DISSERTATION

Internet-Scale Push Systems for Information Distribution—Architecture, Components, and Communication

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften
unter der Leitung von

o.Univ.Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
E 184-1
Institut für Informationssysteme

eingereicht an der

Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Dipl.-Ing. Manfred Hauswirth
Matr. Nr. 8420623
Kunzgasse 7/15, 1200 Wien

Wien, im August 1999 _____

Kurzfassung

Diese Dissertation präsentiert ein architekturelles Modell und eine Referenzimplementierung für Push-Systeme. Push-Systeme kehren das auf Client-Initiative basierende Kommunikationsparadigma (Pull-Modell) des World-Wide Web und der meisten anderen verteilten Systeme um – damit wird Informationsverteilung und Informationsauffindung für den Anwender erleichtert. Im Pull-Modell obliegt es dem Benutzer bzw. dem Client eine Anforderung zu stellen, wenn neue Information benötigt wird. Push-Systeme hingegen ermöglichen asynchrone Verteilung von Information: Sobald dem Benutzerprofil entsprechende Information verfügbar wird, wird sie verteilt. Im Push-Kommunikationsmodell kündigt ein Informationsproduzent die Verfügbarkeit bestimmter Klassen von Information an, interessierte Konsumenten subskribieren bestimmte Informationsklassen und der Informationsproduzent veröffentlicht in regelmäßigen Abständen seine Informationen (d.h., verteilt sie an die Konsumenten). Dadurch wird das Auffinden von Information erleichtert und die Information wird zeitgerecht verteilt. Durch dieses Kommunikationsmuster entstehen komplexe Problemstellungen, die für eine breite Anwendung von Push-Systemen bewältigt werden müssen: Das System muß in bezug auf hohe Benutzerzahlen und die dafür benötigte Netzwerkbandbreite skalieren und zufriedenstellende Verzögerungszeiten für zeitgerechte Benachrichtigung über die Verfügbarkeit von neuer Information besitzen; die verteilte Information muß authentifiziert werden können und ihre Integrität muß gewährleistet sein; des weiteren sollten elektronische Zahlungsmethoden und Geschäftsmodelle unterstützt werden. Derzeit verfügbare Systeme bieten nur teilweise Lösungen für diese Bereiche: Die meisten verfügbaren Push-Systeme basieren intern auf dem Pull-Modell und fragen in regelmäßigen Abständen die ihnen bekannten Informationsproduzenten nach neuer Information ab, wodurch das Push-Konzept ad absurdum geführt wird; häufig ist die Skalierbarkeit der Systeme begrenzt; nur wenige stellen die Möglichkeit einer Authentifizierung und Integritätsprüfung der verteilten Information zur Verfügung; und von keinem der derzeitigen Systeme werden elektronische Zahlungsmethoden und Geschäftsmodelle adäquat unterstützt.

Diese Dissertation definiert und beschreibt ein Kommunikations- und Komponentenmodell für Push-Systeme. Das Kommunikationsmodell wird dem Client-Server-Modell und ereignis-basierten Systemen gegenübergestellt und als gleichwertiges Kommunikationsmodell für eine Klasse verteilter Systeme identifiziert. Das Komponentenmodell bietet einen Rahmen für den Vergleich und die Bewertung von Push-Systemen und den ihnen zugrunde liegenden Designentscheidungen. Das Komponentenmodell besteht aus folgenden Komponenten: Produzent, Konsument, Versender, Kanal und Transportsystem. Dieses Modell kann auch als Ausgangsbasis für die Entwicklung einer Push-System-Referenzimplementierung verwendet werden. Der zweite Teil dieser Dissertation präsentiert eine solche Referenzimplementierung mit Namen Minstrel, die das Komponentenmodell als Architektur für die Entwicklung austauschbarer Komponenten verwendet und eine offene Protokoll-Suite zur Informationsverteilung auf Internet-Größenordnung definiert.

Minstrel ist eine vollständig auf Java basierende “Proof-of-Concept”-Implementierung unter besonderer Berücksichtigung der Problembereiche Skalierbarkeit, Authentifizierung und Bezahlung. Skalierbarkeit wird durch ein hierarchisches Transportsystem erreicht, das für Produzenten und Konsumenten konzeptuell transparent ist. Um Konsumenten zeitgerecht über die Verfügbar-

keit neuer Information zu benachrichtigen, wird eine aktive, hybride Verteilungsstrategie verwendet, die sich nicht auf spezielle Multicasting-Infrastrukturen stützt. Die Verteilungsstrategie und die dabei verwendeten Protokolle werden im Detail präsentiert und evaluiert. Erste Auswertungen der Verteilungsstrategie in bezug auf Skalierbarkeit und Verzögerungen sind vielversprechend. Für die Unterstützung von elektronischen Bezahlungsmethoden und Geschäftsmodellen stellt Minstrel ein flexibles und generisches Modell zur Verfügung, das die verwendeten Geschäftsmodelle von den darunterliegenden elektronischen Bezahlungsmethoden entkoppelt. Dadurch unterstützt Minstrel eine große Bandbreite unterschiedlicher Geschäftsmodelle mit (theoretisch) beliebigen elektronischen Bezahlungsmethoden. Des weiteren bietet Minstrel eine auf dem Konzept der digitalen Unterschrift basierende, verteilte Authentifizierungsinfrastruktur, die die Überprüfung des Ursprungs einer Information ermöglicht und die Integrität der verteilten Informationen gewährleistet. Diese Infrastruktur bietet außerdem einfach zu verwendende Abstraktionen der darunterliegenden, komplexen Sicherheitskonzepte. Minstrel unterstützt sowohl die Verteilung von statischer Information, als auch von ausführbaren Programmen (mobile Java-Programme). Um den Konsumenten vor möglichen Sicherheitsrisiken durch diese Programme zu schützen, stellt Minstrel ein flexibles und weitreichend konfigurierbares Sicherheitsmodul zur Verfügung, das fortschrittliche Funktionalitäten wie die Definition subtraktiver Sicherheitsstrategien und Verhandlung von Sicherheitsaspekten zur Programmlaufzeit bietet.

Abstract

This dissertation presents an architectural model and a reference implementation for push systems. Push systems reverse the pull-based communication paradigm on the world-wide web and in most other distributed systems to support easier information dissemination and discovery for users. The pull model requires the user to issue a request whenever information is needed, whereas push systems support asynchronous information distribution: Whenever information of the user's choice becomes available, it gets distributed. In the push communication model, an information producer announces the availability of certain types of information, an interested consumer subscribes to this information, and the producer periodically publishes the information (pushes it to the consumer). This simplifies the discovery of information and provides timely information dissemination but introduces complex problems that challenge the widespread deployment of push systems: scalability to large numbers of users in terms of network bandwidth, timely notification of information availability, authenticity and integrity of information, and support for payment methods and business models. Current systems fall short in addressing these issues. Most available push systems actually use a pull-based distribution approach where clients check for new information at configurable intervals; frequently scalability is limited, many systems lack services to provide information authenticity and integrity, and moreover, the important issue of payment models is not adequately addressed by any existing system.

This thesis defines and presents a communication and component model for push systems: The communication model contrasts push systems with client-server and event-based systems; the component model provides a framework for comparison and evaluation of different push systems and their design alternatives. The component model consists of producers and consumers, broadcasters and channels, and a transport system. It can also be used as basis for developing a reference implementation for push systems. The second part of the thesis presents such a reference implementation called Minstrel where the component model is used as an architecture for developing plug-compatible components and to devise an open protocol suite for Internet-scale content distribution. Minstrel is designed as a Java-based proof-of-concept implementation of the architectural model and addresses the issues of scalability, notification, authenticity, and payment highlighted above. To provide scalability, Minstrel uses a hierarchical transport system transparent to both producers and consumers. For timely notification of information availability it employs an active, hybrid broadcasting strategy that does not rely on special multicast infrastructures. Minstrel's distribution process and protocols are presented and evaluated in detail. First evaluations of the broadcasting strategy are promising in terms of scalability and delays. To support payment and business models, Minstrel provides a flexible and generic payment model which decouples the business model employed from the underlying payment method(s), so that it can be used for a variety of business models with (theoretically) arbitrary payment methods. Minstrel includes a distributed authentication infrastructure that facilitates authentication of information origin and integrity checks through digital signatures and offers high-level security abstractions. Minstrel supports the distribution of static and executable content (mobile Java code). To protect consumers from malicious mobile code, Minstrel provides a highly configurable Java secure execution framework that offers advanced features such as subtractive security policies and runtime security negotiation.

Acknowledgments

Several people have helped and supported me in writing this dissertation.

First of all I would like to thank my girl-friend Sabrina for her patience and mental support.

Thanks to Mehdi Jazayeri who has encouraged me to pursue this work.

Thanks to Wolfgang Lugmayr for our fruitful discussions on business cases and data structures.

Thanks to Harald Gall for our discussions on several central aspects of this work.

Thanks to ρ who read the central parts of this dissertation and provided valuable comments.

Special thanks to the Minstrel team (Michael Fischer, Stefan Jakl, Clemens Kerer, Roman Kurmanowytsch, Michael Pührerfellner, Gerald Spernbauer, Martina Umlauft) for their help in the implementation and our fruitful discussions.

Thanks to Pedrick Moore who proof-read this dissertation and improved my style of writing.

Thanks to all my colleagues at the Distributed Systems Group who took over a lot of my teaching duties and administrative work-load in the last semester. This gave me time to finish this dissertation.

And—last, but not least—thanks to Jethro Tull and their music which has accompanied me since my teenager days. Their song *Minstrel in the Gallery* is one of my favorites and partly inspired the name of the Minstrel system.

MINSTREL IN THE GALLERY

The minstrel in the gallery looked down upon the
smiling faces.
He met the gazes – observed the spaces between the
old men's cackle.
He brewed a song of love and hatred – oblique
suggestions – and he waited.
He polarized the pumpkin-eaters – static-humming
panel-beaters – freshly day-glow'd factory cheaters
(salaried and collar-scrubbing).
He titillated men-of-action – belly warming, hands
still rubbing on the parts they never mention.
He pacified the nappy-suffering, infant-bleating
one-line jokers – T.V. documentary makers
(overfed and undertakers).
Sunday paper backgammon players – family-scarred
and women-haters.
Then he called the band down to the stage and he
looked at all the friends he'd made.

The minstrel in the gallery looked down on the
rabbit-run.
And threw away his looking-glass – saw his face in
everyone.

IAN ANDERSON (JETHRO TULL)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The dominant paradigm of communication on the world-wide web and in most distributed systems is the request-reply model. In this model of distributed information systems, a client actively "pulls" information from the server. Ever since the early days of the Internet, systems such as electronic mail and Usenet News have attempted to overcome the deficiencies of this pull model by allowing producers of information to "push" their information closer to the clients. In the push model, an information producer announces the availability of certain types of information, an interested consumer subscribes to this information, and the producer periodically publishes the information (pushes it to the consumer). The pull and push models are contrasted in Figure 1.1.

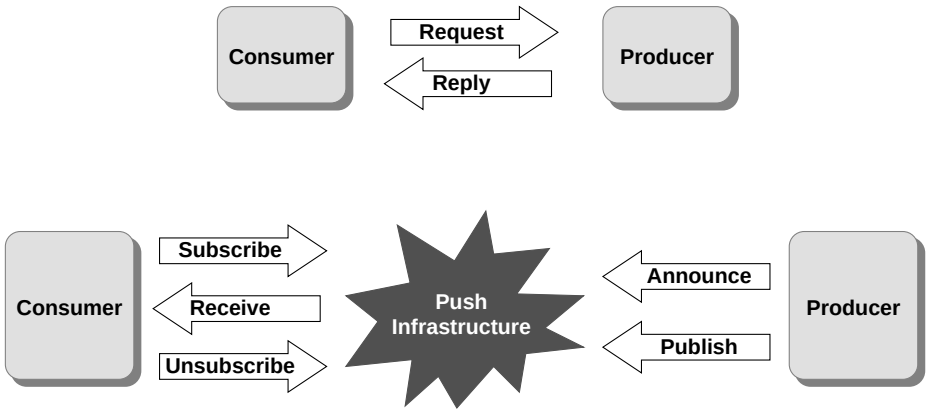

Figure 1.1: Pull vs. push

Several reasons motivate the need for push systems. The most important one is that the WWW is based on a simple request/reply scheme [11, 42] that requires the user to issue a request whenever he/she needs information. This imposes a "synchronous" interaction scheme, whereas push systems allow asynchronous information distribution: Ideally, whenever information of the user's choice becomes available it gets distributed.

1.1 A Scenario

To illustrate the applicability of the push approach and show the various issues and requirements of a push system, this section presents the example of a news agency information system. This scenario exhibits all essential properties of a push system and demonstrates the practical relevance of the issues and requirements discussed in the following sections of this thesis.
Figure 1.2 depicts a typical news agency scenario.

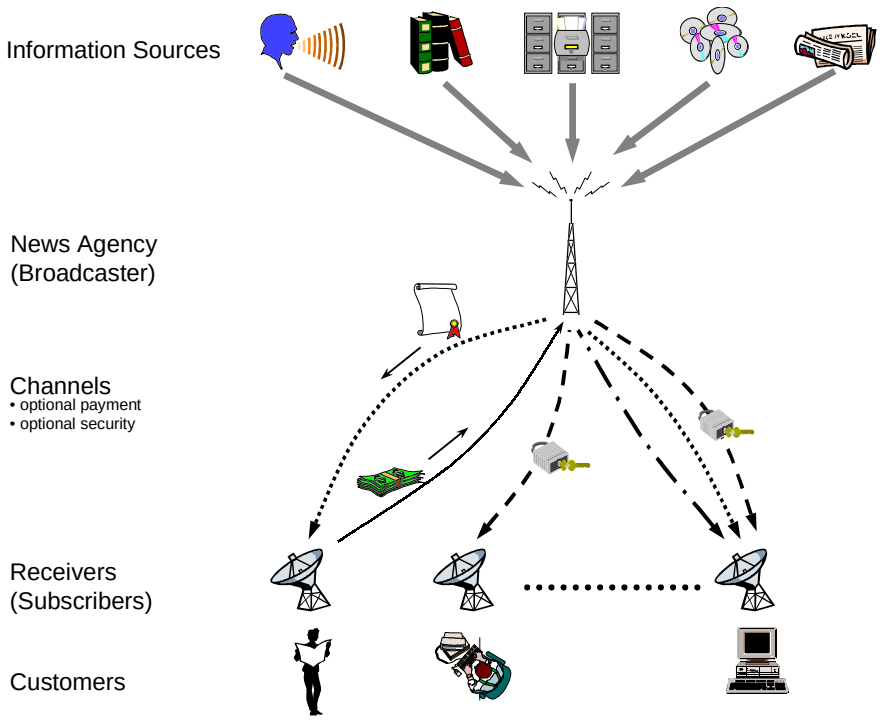

Figure 1.2: News agency scenario

A news agency buys or receives information from various sources, such as reporters, newspapers, databases, libraries, and stock exchanges. It can edit the information, combine it with other information, and classify it. For example, it can combine the stock quotes of a company, which it receives from the stock exchange, with financial analyses or background information about the company. The result of this process is categorized units of information.
These units of information can then be disseminated ("broadcast") via push channels according to their categorizations. Every channel is associated with a defined set of categories so that customers can choose and subscribe channels they are interested in. Typical categories would be local news, national news, financial news, weather, and sports. Every unit of information can be associated with several of these categories, so that the data may get distributed via multiple channels.
At the customer end, users run a special receiver software that facilitates access to the news

agency's channels. The receiver software manages the customer's subscriptions to channels and user profile, and interacts with the news agency's "broadcaster" on behalf of the user. It receives the information sent over the subscribed channels and provides a user interface that allows the customer to access this information.

Because the news agency offers many different channels and feeds information to many customers, its distribution infrastructure must scale well to large numbers of users and provide for timely delivery while imposing only minimal additional constraints. Specifically, this infrastructure must be as transparent as possible towards broadcasters and receivers. While this is a major component of every Internet-scale push system, it is omitted in Figure 1.2 for reasons of simplicity, but will of course be discussed in detail in later sections.

It is important to note another simplification in the scenario of Figure 1.2: Receivers interact with only one broadcaster. In a real world setting, however, receivers can subscribe to multiple channels from multiple broadcasters (news agencies or other push providers).

So far the scenario provides for the timely, topic-based dissemination of news. A news agency, however, has to fulfill additional constraints because its business depends upon a high level of data confidentiality. Receivers want to be able to rely on the information and possibly use it as a basis for business decisions, so proof of origin (data authenticity) and proof that the information has not been tampered with (data integrity) are very important. These proofs also provide the foundation for limitation of legal liability and must be provided by the underlying infrastructure.

Since the news agency wants to make revenue out of its business, it must charge for the information it disseminates over the channels (a minor fraction of information/channels will, however, be free of charge). Thus the push system must support a range of payment methods and business models, such as flat fees or pay-by-view.

If the news agency wants to protect intellectual property rights or simply prohibit unauthorized access, channels must be encrypted ("locked"). However, only important channels will be secured in this way, since encryption always leads to additional delays and higher costs.

The news agency is an information distributor rather than an information generator. It must either convert the data it collects from different sources to a few standard types or else leave it to the receivers to deal with the various data formats. The first option is unnecessary and infeasible for the rapidly growing number of Internet content types. Therefore the underlying push system must support the dissemination of arbitrary content types in a way similar to the world-wide web. This is referred to as being "transparent" towards the transported information. The receiver software, however, must be able to deal with the received data in a meaningful way.

Content types fall into one of three categories:

Static: static, immutable data like HTML text, pictures, audio files, etc.

Executable: code (plus accompanying data) that is intended for execution at the receiver's site

Streaming: data that exhibits real-time characteristics, for example real-time audio or video; streaming content is not received as a whole before it is "played" or "visualized"; instead a continuous data stream requiring real-time distribution is established between the sender and the receiver, and the receiver processes the received data continuously (e.g., plays a piece of music)

The news agency would use a push system to distribute static and executable data. Like the world-wide web, push systems do not support streaming data. This content type is beyond their application domains and requires specialized infrastructures, such as [141].
Executable content can be used in several ways: to provide client-side processing (for example, an interactive questionnaire) or to enhance the capabilities of the receiver (for example, software updates of the receiver or code to support new content types). This, however, raises additional security issues for the receiver software. Received code must be authenticated and verified and be executed in an environment that protects the receiver's site from malicious code.
The issues presented in the above scenario define the main requirements that must be fulfilled by every Internet-scale push system: scalability to large numbers of users without affecting timely distribution, provision of authenticated and secure information, support for a wide range of content types, support for mobile code (including an infrastructure that deals with the security issues involved), and support for e-commerce by facilitating the use and implementation of flexible payment methods and business models.
This thesis will address all these problem areas in detail.

1.2 Key Issues and Application Domains

Some of the problems of distributed information systems that can be solved with the push model are:

Discovery of information: With the vast amount of data available on the Web, users have difficulty finding the desired information. Even though good search engines exist, the quality of information found is still proportional to the user's knowledge and skills (keyword selection, intuition). With push systems, special channels can be used to announce the availability of information.

Timeliness of data: With web-based systems, the client must continually poll to ensure that its data is up to date. With push systems, the client may be notified as soon as data is updated on the server. In such a scheme, a push system may use the services of an event-based infrastructure.

Authentication of information: Validating the identity of the information provider and the identity and integrity of the information itself is simplified in push systems because of the subscription phase of the interaction. At this point, encryption keys may be exchanged that will later be used to authenticate the server.

User customization: With push systems the user can state his/her preferences explicitly. Thus it is easy to provide information that is focused on the user's interests. The user's requirements on the data and its properties, e.g., data format, priority, keywords, can be enforced before the data is delivered.

Provider-side information tailoring: Not only the user can customize the data and its properties but also the provider. The provider can control when the user sees which information.

This decision can be based on user profiles provided by subscription information, interest analysis, etc., and allows information to be tailored to the user's interests. This functionality can also be exploited for advertising if allowed by the user.

Traffic reduction: Push systems may facilitate reduction of network traffic. Users trying to locate information may cause heavy traffic, but since push systems can tailor data to users' profiles, unnecessary requests and thus traffic may be decreased considerably. Additionally, an appropriate transport infrastructure can further cut down on network bandwidth, for example, by using repeaters that are collocated with Internet service providers.

Detached and mobile operations: Recently Personal Digital Assistants (PDAs) have become very popular and gained widespread use. PDAs raise new challenges for distributed information systems, since they are mobile and not always connected. By default, push systems address most of these issues. With push systems, PDAs can be loaded with (channel) information, be used offline, and at a later time be updated when connected again.

Numerous application domains have already been devised for push systems [61]. To show the wide applicability of the push model as a paradigm for building distributed applications, here several distinct applications are outlined whose design can be decomposed in terms of push concepts.

Intra-company employee information systems: Many organizations have proprietary and ad hoc systems for keeping their employees informed about their organizational news. This is sometimes viewed as one of an organization's most important and most difficult tasks. Such a system may be built as a standard push system.

Electronic maintenance manuals: Companies that produce appliances have maintenance manuals that are carried by their maintenance workers when they are called to repair appliances on site. The updating of such printed manuals is costly and tedious. With a push system, each product line could be associated with one channel and maintenance workers could subscribe to whatever channels needed.

Stock ticker system: This is a classic example of event-based and push systems.

Electronic newspapers: Currently electronic newspapers are typically provided as web sites which users visit at regular intervals. A push system more closely resembles the real-world newspaper model by allowing the user to subscribe to a set of topic-based information channels (weather, sports, local news, world news, etc.) and receive the information at his/her computer as soon as it gets distributed. This is in effect a personalized newspaper.

Tourism information systems: Travel agencies and other tourism-related businesses depend on fast, wide-reaching announcement facilities to advertise destinations, hotels, tours, special offers, and other tourism products. A web site may be too "passive" for this purpose. A push system could be used to support active advertisement with booking functionalities. Every group of offers or products, such as last minute bookings, could be associated with a channel, and customers could subscribe to these channels according to their interests.

Distance education: With a push system, every course offered is associated with a channel through which course material is distributed. Students subscribe to the courses they want to take and automatically receive the course material as soon as it becomes available.

Software distribution: A push system can send subscribers new software or an update as soon as it becomes available. If supported by an appropriate software deployment infrastructure at the client site, installation and maintenance of software can be automated.

News agency information systems: News agencies sell various kinds of information. Timeliness of dissemination and authenticity of information are prime issues. Powerful classification and filtering mechanisms are needed that allow customers to select information of interest from the huge amounts of data a news agency offers. Such a system can be modeled as a standard push system.

The existence of such diverse applications, all of which can be designed as specific instances of push-based systems, speaks for the inherent utility of the push model and concepts. In all of these systems, one can easily identify distinct producers and consumers, and also the necessity of a subscription phase.

1.3 Pedigree of Push Systems

A look at the historical development of the software around the Internet provides a good understanding of the roots and rationale of push systems. Almost immediately after the introduction of the ARPANET, the predecessor to the Internet which provided a hardware link among a dozen computers, its potential as a medium for communication among people was recognized. Electronic mail and Usenet News were the first two attempts to address this challenge. The world-wide web has grown from these basic forms of communication.

The differences between these communication tools are the number of participants involved, whether the sender needs to know the identity of the receiver(s), and the amount of structure in the interaction. With email, the sender actively sends (pushes) the information to known participants who are notified of the arrival of the data. With News, the sender posts the data to a common area and anonymous readers are expected to check the common area regularly for new information. The messages sent by email and News remain unchanged once posted. In web communication, the data passively waits at the sender's home site to be "pulled" by receivers. While it waits, it may undergo changes: The document is said to be "live."

The essential problems to communication among the users of such distributed systems are the discovery of the availability of information, the notification of the existence of new or updated information, the maintenance of consistent state among senders and receivers, the limitations on forms of supported interaction patterns, limited security and authentication, and most importantly, the scalability of the system. Many systems have emerged around the Internet to solve one or more of these problems.

The first solutions for coping with the huge amount of information available on the world-wide web were bookmarks, hot-lists, and user-maintained "jump-stations." But these tools failed to

keep pace with the rate of information expansion on the web. An automated scheme for gathering index information and providing search facilities became necessary. This led to the introduction of **search engines**: Usually stationary **robots** gather index information by recursively retrieving documents and indexing them.[1] This index data can then be searched via a user interface. At present there are several competing search engines, e.g., AltaVista [2], Excite [41]. This approach, however, has its problems: Robots can cause high server and network loads, indexes are not exchanged between the search engines, which leads to inefficient multiple indexing runs for the same data, and there is still much information that cannot conveniently be found by average users because the data is not or only poorly structured.

Thus **portal sites** like *Yahoo!* [183] or *Netscape Netcenter* [126] were introduced as a supplement to search engines.[2] Portals try to give a structured view on the data available and allow the user a certain degree of freedom to get a customized view on the data. Typical examples here are *My Yahoo!* [182] and *My Netscape*[3] [123]. The issue of notification, however, still remains open.

The combination of electronic mail and the world-wide web offered by several portal sites, such as Netscape's In-Box Direct service [122], provides an interesting push-system-like approach: Users sign up with a mailing list and receive mails in regular intervals; these mails typically contain an HTML document that is displayed as in a browser when read with an appropriate mail tool. The HTML document contains links that the user can click on to retrieve the corresponding documents. This approach is discussed further and contrasted with the push approach in Section 3.2.3.

Growing demand for solutions to the issues listed above created a big rush towards **push systems** in late 1996. Push systems reverse the communication pattern of Internet-scale information systems by actively disseminating information to consumers. Consumers, instead of having to check for new information repeatedly, subscribe to information channels and receive updated information as soon as it becomes available. Sections 3.3 and 3.4 examine and compare available push systems. Interestingly, the first push system approach was introduced as early as 1992 by the *dynamic document* concept of Netscape Navigator 1.1 [121]. Its basic ideas were *server push* and *client pull*. With server push, the server sends data which is displayed by the browser, but the connection between server and client remains open. Later the server may continue to send other pieces of data to the client. Client pull automates reloads: The server sends data which includes a `Refresh` directive specifying a time delay and a URL in the HTTP response or in the document header. After the given delay, the client loads the document specified by the URL.

Nearly at the same time that push systems were coming into use, Internet-scale **event-based systems** were introduced in academia. They are closely related to push systems but have not yet gained widespread use in industry. In contrast to push systems they focus on notification and event distribution aspects. A comparison between push systems and event-based systems to clearly identify their relationship is given in Section 3.5.1.

[1] A few systems allow sites to provide their own index data, e.g., Harvest [12].

[2] Actually they partly evolved in parallel. Now many portal sites have an integrated search engine, and vice versa, many search engines also offer a web portal.

[3] Recently Netscape extended the customization options: Users can setup a Rich Site Summary (RSS) file that facilitates the definition of an additional customized information aggregation ("channel") beyond the predefined categories [124]. Its capabilities, however, are rudimentary.

1.4 Contribution of the Thesis

This thesis describes and analyzes the concepts and issues of Internet-scale content distribution with respect to push systems. It defines and presents the *first general communication and component model for push systems*. The model, which consists of producers and consumers, broadcasters and channels, and a transport system, can be used to accurately describe the structure of existing push systems. Furthermore it provides a classification framework for analyzing and comparing the important design decisions in those systems and it compares prominent push systems according to this framework.
The model can also be used as basis for developing a reference implementation for push systems. The *Minstrel push system* presented in the second part of this thesis is such a *reference implementation* where the component model is used as an architecture for developing plug-compatible components and to devise an *open protocol suite for Internet-scale content distribution*. It is designed as a Java-based proof-of-concept implementation of the architectural model and serves as an *extensible platform* for further research in the push area. Minstrel's main goals are *scalability*, *active push*, *authenticity and integrity of content*, and *flexible support for payment methods and business models*.
In Minstrel the interacting parties—information producers and consumers—are clearly separated by a *hierarchical transport system* that is *transparent* to both parties. This transport system facilitates *scalability to large numbers of users* while minimizing resource consumption and network traffic, and distributing computational load.
In contrast to other Internet-scale push systems that use client-side polling at configurable intervals, Minstrel employs an *active, hybrid broadcasting strategy*. This provides timely notification of information availability and requires no special multicast infrastructure. First analyses of the broadcasting strategy are promising in terms of scalability and delays.
A main goal of Minstrel is to offer a *platform for information commerce* over the Internet. The two main requirements Minstrel must address to be applicable in a business environment are authenticity and integrity of information, and support for payment methods and business models. Minstrel's *distributed client-server authentication infrastructure* facilitates authentication of information origin and integrity checks through digital signatures (based on public key encryption and certificates). The infrastructure also provides *high-level security abstractions* that simplify its usability by encapsulating the low-level security details.
To support payment and business models, Minstrel offers a *flexible and generic payment model* that can be used for a variety of business models, such as pay-per-view or volume-based. It decouples the business model employed from the underlying payment method(s), so that (theoretically) arbitrary payment methods can be used. The model is evaluated using the Millicent micro-payment protocol.
Minstrel supports the distribution of *static and executable content*. Executable content, i.e., mobile code, is executed at the receiving site and can also be used for extending the capabilities of client-side Minstrel components. Since mobile code can threaten the security and system integrity of the client, Minstrel includes a *highly configurable secure execution framework for Java code*, which offers advanced features such as subtractive security policies and (interactive) runtime security negotiation.

1.5 Organization of the Thesis

This thesis is structured as follows. Chapter 2 presents the Communication and Component Model for Push Systems. Although several push systems have gained widespread use on the Internet, no such model exists so far. Based on this model, several prominent push systems are compared in Chapter 3. A structural classification is given that compares these systems in terms of the main features described in the previous chapter. To adequately set the field, related paradigms, such as event-based systems and mobile code, are also addressed. Chapter 3 also discusses other push approaches that have not gained widespread application, and multicast infrastructures as a related base technology.

Chapters 4 through 6 present the Minstrel push system in detail. Chapter 4 provides an overview of Minstrel's architecture and describes the distribution strategy it employs. Its protocols are presented in detail and evaluated on the basis of a representative scenario. Chapter 5 then describes the core components of the Minstrel system. These are derived from the component model in Section 2.2 by using it as an architecture for developing plug-compatible components for push systems. Chapter 6 rounds out the description of Minstrel by explaining how Minstrel addresses the issues of information authentication and integrity, mobile code security, and electronic commerce and payment. The problems in these domains are presented, Minstrel's proposed solutions are discussed, and their implementations in Minstrel are showcased.

Chapter 7 gives an evaluation of the Minstrel push system as an instantiation of the model of Chapter 2 and summarizes the previous chapters. An overview of future work rounds out the thesis.

Chapter 2

A Communication and Component Model for Push Systems

This chapter presents a communication and component model for push systems. Surprisingly, despite the widespread use of many push services on the Internet, no such models exist. The communication model in Section 2.1 contrasts push systems with client-server and event-based systems and analyzes their impact on scalability, network load, and state maintenance. The component model presented in Section 2.2 provides a basis for comparison and evaluation of different push systems and their design alternatives. The component model consists of producers and consumers, broadcasters and channels, and a transport system. The concerns of each of these components are explored in detail. Several prominent push systems will be compared in the next chapter using this component model. Section 2.3 discusses a number of open issues that challenge the widespread deployment of push or any other systems on an Internet-wide scale. Payment models are the most important among these, as they are not adequately addressed by any existing system. Thus a preview of the payment approach of the Minstrel system is also given which will be described in detail a later chapter. Finally, we summarize and give our conclusions in Section 2.4.
The described model was presented at the European Software Engineering Conference (ESEC) [69].

2.1 A Comparison of Distributed Communication Models

A distributed system consists of several computing nodes connected by a computer network that provides for communication among the nodes. When applied in a distributed environment, the traditional task of software decomposition must also deal with allocating (or mapping) software modules to the different nodes. One of the performance goals of distributed software design is to minimize the amount of communication needed among the nodes. Less communication leads to higher scalability, that is, the ability of the system to support more nodes or more users.
The *client-server model* provides an architectural approach for organizing the software for distributed platforms. The model assumes a small number of servers (say, 10) and a moderate

number of clients (say, 1000). The basic scheme is that clients interact with (human) users and contact the servers to ask for (computationally-intensive or data-intensive) services. The communication model of client-server systems may be called *session-based* (or *stateful*). During a session a client and a server share a state, which is modified through one or more of their interactions.

The popularity of this model has led to its standardization in software environments such as DCE and CORBA, which define standard components to ease the building of such client-server software [175]. Currently popular N-tier architectures are extensions of this basic model that attempt to remedy some of its shortcomings.

The emergence of the Internet and its use as a platform for distributed applications, however, exposed the weaknesses of the session-based communication model in terms of scalability. Internet applications must scale to millions of nodes and users. The primary impediment to scalability is the participants' need to maintain a shared state. Consequently, in the interest of scalability, the world-wide web adopts a *stateless* approach to client-server communication (*web-based model*). In this scheme, each interaction between the client and the server is independent of the other interactions. No "permanent" connection is established between the client and the server and the server maintains no state information about the clients. While this scheme helps scalability, it becomes difficult to maintain a state: The client, the server, or both must maintain the state and ensure its coherence [9]. Web-based applications scale to 1000s of servers and 1,000,000s of clients. Depending on the requirements of an application, the application designer may choose between these two models in client-server computing. The primary tradeoffs are between loose or tight coupling and maintaining state at the client or the server.

The client-server model deals with two participants in the communication. In the *peer-to-peer* model, the application is decomposed among many peer nodes instead of clients and servers. For this reason, the nodes here are referred to as producers and consumers rather than clients and servers. In the peer-to-peer model, communication begins with a subscription phase in which a consumer registers its interest with a producer. At this point, the peer-to-peer model may be divided also into two subclasses: the event-based and the push-based models.

In the *event-based model*, nodes are loosely-connected and behave symmetrically: Any node may produce events and any node may consume events. This model scales to many producers and many consumers because there is no coupling between them. This model has recently received considerable attention [179].

The *communication model of push-based systems*, on the other hand, is tightly coupled and asymmetric: Certain nodes are designated as producers and others as consumers. In contrast with the event-based model, push-based systems scale to fewer producers but more consumers. They may be viewed as a specialization of the event-based systems with designated producers and consumers and channels to connect each producer with interested consumers. Dissemination in push-based systems is done on the basis of particular channels rather than event classes as in event-based systems. The use of channels for information classification is a major distinction from event-based systems. Channels increase the coupling but improve the performance in certain situations.

Table 2.1 contrasts the four communication models described above in terms of the primary tradeoff between coupling and scalability.

	Client-server		Peer-to-peer	
	Session-based	Web-based	Event-based	Push-based
Coupling	tight	loose	very loose	medium
# of clients	moderate (1000)	high (1,000,000)	many (100,000)	many (100,000)
# of servers	few (10)	many (100,000)	many (100,000)	few (100)

Table 2.1: Comparison of communication models

The four communication models of distributed systems occupy four areas in the design space of distributed systems. Figure 2.1 gives a rough view of the design space plotted along coupling and scalability axes.

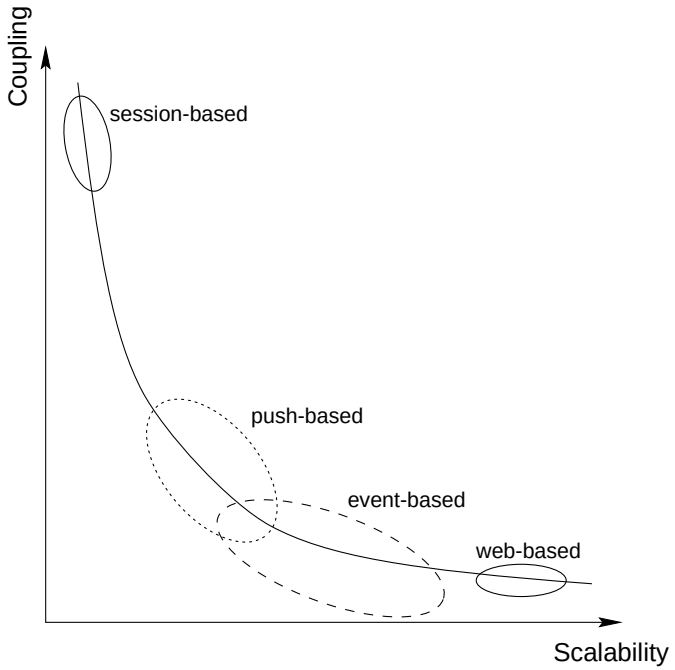

Figure 2.1: Degree of coupling vs. degree of scalability

Session-based and web-based models have been the subject to many studies. Event-based systems on the Internet-scale have also recently been subject of intensive analysis [144, 179]. Push systems, on the other hand, seem to have had a short time in the limelight and then fallen out of favor in the academic world. They are, however, heavily used in practical applications on the web. The purpose of this chapter is to establish the push-based model as a viable model of distributed applications.

2.2 A Component Model for Push Systems

This section presents a component model for push systems which has been derived from an analysis of existing push systems.
In its simplest form, a push system consists of producers and consumers of information that are connected through channels. A consumer (receiver) subscribes to a channel and receives any information that a producer (information source) sends through it. Thus the connection phase of client-server systems is replaced by an early subscription phase.
In practice, a broadcaster component is used to separate the concerns of channel handling from the information source. A broadcaster is responsible for managing channels and sending information along them. Thus the broadcaster controls and schedules the dissemination process and distributes data via a set of channels to the consumers.
An information source feeds information to a broadcaster together with rules on how and where (on which channel) to distribute this data. The broadcaster may apply filters to the data before disseminating the data (via channels) to consumers that have subscribed to receive the content of certain channels. Receivers may apply filters, too, and accept content only if it passes through them. To provide scalability to large numbers of users, the distribution process involves a transport system which is conceptually transparent for broadcasters, receivers, and channels.
The transport system consists of caches, repeaters, and proxies. These components can cooperate via specialized transport system protocols that are internal to the transport system and not visible outside. Caches and repeaters are useful for conserving network bandwidth by reducing the load on broadcasters and bringing channel data "closer" to the receivers. In the case of a cache, this is done on demand, whereas a repeater is preloaded. Proxies model situations where no direct connection to a push system component is possible, e.g., for security reasons or to control network traffic. The proxy component thus acts on behalf of some other components.
Figure 2.2 depicts the component model of a push system and Figure 2.3 shows a sample collaboration (UML sequence diagram) between the components of the model.
The information source provides new data for a specific channel to the broadcaster. The broadcaster applies filters to the data to limit data transfers and sends the data (in parallel or iteratively) to the set of repeaters (for scalability reasons) for which the filters succeeded. The repeaters then redistribute the data to the actual subscribers (receivers). For higher scalability, additional levels of repeaters may be necessary.
Every broadcaster can send to multiple channels and every receiver can receive from multiple channels. In Figure 2.2 some of the arcs representing channels and backchannels cut through components of the transport system to show that these components are necessary for scalability purposes but are transparent to the channels and the dissemination process.
Having decomposed a push system into the components in Figure 2.2, we will now take a detailed look at the concerns of each component of the component model.

2.2.1 Channel

A channel is a (logical) connector between a broadcaster and a receiver. It determines the protocols between these components. The most important of these protocols are the channel access

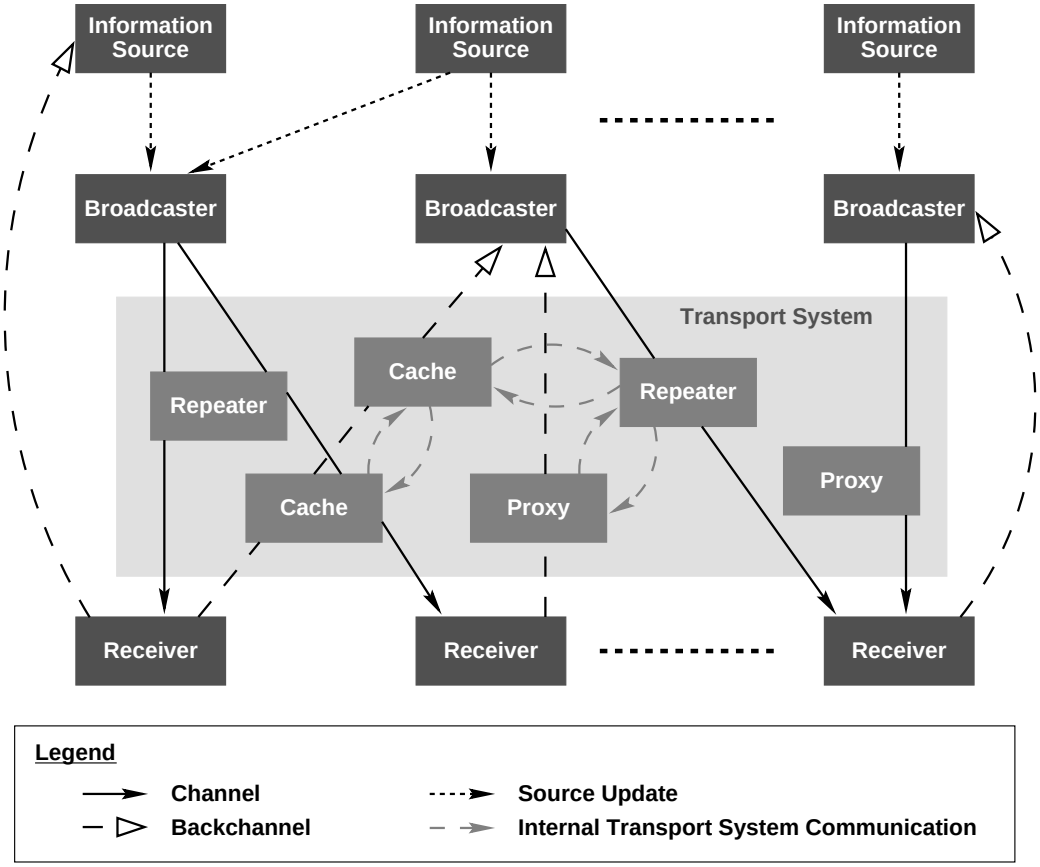

Figure 2.2: Components of a push system

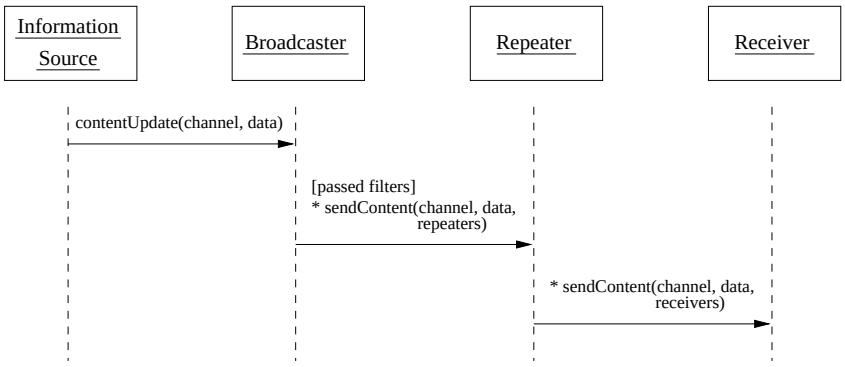

Figure 2.3: Content distribution via a channel

protocol and the subscription protocol. Channels provide for many-to-many connections among broadcasters and receivers: Each broadcaster provides a set of channels that receivers can subscribe to; each receiver subscribes to a set of channels. Channels are a major distinction between event-based systems and push systems. The channel concept of push systems already provides a coarse level of information classification that event-based systems usually lack. When data is grouped according to information type, the total amount of data transfers can easily be easily because data (events) need only be distributed to channel subscribers. Additionally, finer filtering can be applied to the contents of a channel (as in event-based systems).
A channel determines several properties of the data to be disseminated and the supported functionalities:
Type of information: the focus of the data that is distributed in a channel (e.g., financial news, weather forecasts, software updates).
Data format: the formats (e.g., HTML) and semantics (e.g., static, executable) of a channel's data. Static data means text files, pictures, data, etc., whereas dynamic content refers to executable programs, etc.
Personalizing/filtering: This property determines the extent of user customization that is possible (e.g., content selection, operation modes, payment).
Content expiration: Channel content can be transient or persistent. An expiration strategy for the channel must exist to prevent using up the consumer's resources.
Update strategy: This defines how updates of the channel's contents are done. Possible strategies are replacement, incremental, or differential updates (depending greatly on the type of information and the data format). The timing of updates has impact on data accuracy, network traffic, and scalability.
Scheduling strategy: The main scheduling options are time-scheduled versus content-scheduled. Time-scheduled channels deliver "unrepeatable," "live" content depending on the access time of a channel (comparable to TV or radio). Content-scheduled channels deliver content independent of real time.
Operation mode: Consumers may not be online all the time. Support for offline/mobile operation with feasible synchronization protocols is necessary.
Payment: Certain channels (support channels, special contents, etc.) may involve payment. The channel configuration determines which business model to use: pay-per-view, content-based, time-based, flat fee, etc.
Some of the properties described above can be modeled by viewing a channel as a data stream that is directed through a set of configurable filter components, as shown in Figure 2.4. Filtering can be done at both the producer end and the consumer end.

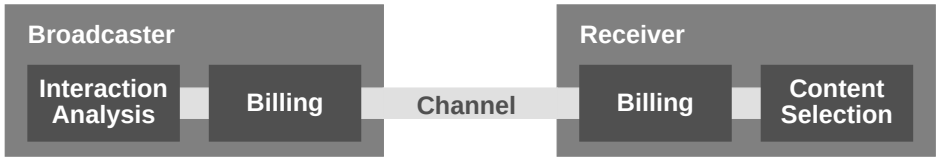

Figure 2.4: Filtered data-stream view of a channel

Channels model a 1:n relationship between a producer and its consumers. Additionally, a consumer can communicate information back to a broadcaster or information source via a backchannel. This“up-stream” communication is a 1:1 relationship between a consumer and a producer. It is usually done in a client-server style and thus is conceptually “outside” of push systems. Frequently a lighter version of the communication facilities or a different medium is used; a software update channel, for example, may have an HTTP-based backchannel. Nevertheless, the more closely and seamlessly a channel and backchannel are integrated, the better. Backchannels can exist on a per-channel basis, for a set of related channels, or for the full set of channels available from one producer.
Additional channel properties are given in [180].

2.2.2 Broadcaster

A push system has at least one broadcaster component that offers channels and distributes data to the subscribers of the channels. For small-scale intranet applications, one dedicated broadcaster may suffice. For large-scale applications that provide channels to thousands of subscribers a single component will not do. Scalability requires a specialized broadcasting infrastructure.
The broadcaster itself may be distributed. A set of broadcasters may provide the channels and exchange updates among themselves to stay in sync. The broadcaster may be organized according to a standard distributed data management scheme such as primary copy replication or data partitioning [22].
Typical configurations include:
Primary broadcaster: One broadcaster is the single source of the channel data and distributes information to a hierarchically organized set of other broadcasters (repeaters). Repeaters do not add information to a channel. They serve solely as a means of bringing data “closer” to consumers and achieving scalability.
Partitioned broadcasters: A set of active broadcasters provides parts of the data and functionality. To the consumers, however, this system provides the illusion of a single entity. Internally, a synchronization strategy must be used. Such architectures can vary considerably in the type and degree of distribution and add another magnitude of complexity.
Simple broadcasting: The broadcasting system does not actively manage the transport and distribution of channel contents but relies fully on functionalities of the transport system. The simplest pattern is a single broadcaster that relies on a caching infrastructure inside the transport system. Most currently available products follow this approach. Although it may seem unrealistic, it has already been applied successfully by other systems like the world-wide web.
Real systems may be combinations of these approaches. The primary goal is to enable receivers to access channels from a broadcasting component that is “close” to them in some respect (bandwidth, delay, etc.) to minimize network traffic, reduce delays, and allow scalable systems.
See [26] for additional discussions on strategies of distributed event dispatching that can also be mapped to push systems.

2.2.2.1 The Notion of Broadcasting

So far the notion of broadcasting in the context of push systems has only been used informally. Broadcasting in a large-scale push system cannot rely on a medium that offers broadcasting per se (e.g., an Ethernet LAN). For Internet-scale push systems there is no broadcast address concept that could be exploited. Yet the scalability of a push system is in fact determined by the broadcasting strategy. The broadcasting strategy tries to balance the tradeoffs between reducing network load and reducing user response time: The broadcaster can reduce user response time if it pushes the data to the receiver node before the user accesses the data, but if the user does not access the data, network bandwidth has been lost. Several standard techniques may be used to implement a broadcasting strategy.

Multicast. Push systems can exploit existing multicast infrastructures (e.g., MBone [87]) and protocols (e.g., RTP [150], NSTP [30]). This greatly simplifies the architecture and implementation of push systems and has several efficiency benefits. However, these resources are accessible by only a limited number of end users.

Client pull. At regular, user-definable intervals, the receiver checks with the broadcaster whether the receiver's view of the channel is still consistent or needs to be updated. This pattern turns the concept of broadcasting upside-down: The initiative changes from producer-side to consumer-side, which is an apparent contradiction to the notion of push systems. However, most of the available *push* systems actually *pull* at the dissemination infrastructure level. With the client pull scheme, complete data accuracy cannot be achieved. High data accuracy and data freshness ("immediate" notification) can only be achieved at the cost of high pulling frequencies, which produce high network traffic and possibly a large number of unnecessary messages if the channel data does not change frequently. Consistency requirements of some channels, on the other hand, may be rather relaxed, and pulling interval of anywhere from 10 minutes to a day may suffice. Additionally, messages that are pulled may be rather small (some 100 bytes). The remaining drawback of client-pull techniques is notification (timeliness of data): How can the receiver be notified of high-priority changes that occur during its pulling interval or of situations which require immediate attention? Despite these shortcomings, pulling is frequently used in push systems since it is robust, simple to implement, allows for off-line operation, and scales well to large numbers of subscribers.

Server push. The broadcaster actively sends content to its subscribed receivers. This solves the freshness problem of pulling but opens up new problems. The main issue that appears in several ways is scalability: Contacting receivers sequentially does not scale even for moderate numbers of subscribers. It would be too time-consuming and would leave receivers with different views of channel information depending on their ordinal number in the pushing process. Therefore a specialized transport infrastructure must be used. For example, organizations can announce a dedicated host that receives data from channels and handles further distribution inside the organization's network. Server push broadcasting also requires a directory of subscribers to be contacted. That imposes additional administration since it must be maintained and kept consistent, and it is a single point of failure. For client pull broadcasting, such a directory is only optional. Moreover, receivers may not be online all the time. This coherence problem must be compensated by re-broadcasts, which adds considerably to the broadcaster's load and com-

plexity. Compared to client pull, the scalability of server push is lower—or requires much more effort in the transport system to achieve similar scalability—but provides better consistency and timeliness of the channel data.

Hybrid approaches. Hybrid approaches combine the advantages of server push (freshness, consistency) and client pull (scalability): Consumers are notified of the availability of new data via a push mechanism (small messages) while the client pulls to transfer the actual data (possibly large amounts of data). This approach is taken in the Minstrel project [67] which is described in Chapter 4: The broadcaster pushes a "sample" (description of the available data, a small-size sample of the real data, and administrative data) to the subscribers of a channel; based on this information the consumers may request the actual data as a "shipment" from the broadcaster. A similar approach is already used by several portal sites (such as Netscape's In-Box Direct service [122]): Users sign up with a mailing list and receive mails at regular intervals (push part); these mails typically hold an HTML document that is displayed as in a browser when read with an appropriate mail tool. The HTML document holds links that the user can click on to retrieve the corresponding documents (pull part).

An issue that is closely connected with the broadcasting algorithm is the channel update strategy. The question is how changes in the data are transmitted to the receiver. A plain replacement strategy would cause high network traffic in the face of possibly minimal changes in the data. A better way to deal with updates is the use of a differential/incremental strategy. This conserves network and computing resources and remedies the problems of network partitions and offline operation.

2.2.2.2 The Notion of Subscription

To receive a channel the client usually has to run through a subscription process beforehand. Thus the connection phase of client-server systems is replaced by a subscription phase.

The subscription process usually requires the user to look up a channel directory (channels + descriptions) to select channels, give some personal information, and provide a profile of interests. The channel directory can be made available as a default channel of a push system. Besides content selection the broadcaster may need subscription information to learn about the receiver's destination address, type of network connection, etc.

The subscription may include a negotiation phase to determine the receiver's access point to channels. If only a single broadcaster without a transport system exists, this is accomplished easily. In all other cases, however, the receiver should be directed to its "closest," "fastest," or "best" access point. In a hierarchical transport system, for example, this could be the nearest repeater or cache. This decision can be made dynamically or based on static configurations. Existing protocols, such as the Service Discovery Protocol [75], can also be exploited for this purpose.

Apart from such details as the distribution address, information about the subscriber is not necessary for any technical reasons. Channel suppliers and broadcasters, however, frequently require subscription information for economic purposes. They want to establish long-lasting relationships with their customers and need to fund their channels. This frequently means carrying advertising, as done, for example, by PointCast (see Section 3.3.4). Since pricing of advertise-

ments depends heavily on the range and readership of a medium, such information is important for the information supplier. Whether a push system can provide support for advertising, may determine its commercial success.
Nevertheless, privacy and security of subscribers must be guaranteed in such a setting. Users must be able to specify which parts of their private subscription information can be used or forwarded to other companies. Additionally, it is important to store user information securely by employing appropriate encryption methods.

2.2.3 Receiver

If we disregard the transport medium, the broadcaster and receiver interact directly. The receiver has two main components: channel access and user interface. The receiver is the interface that facilitates interaction between users and channels. A receiver can subscribe to and receive multiple channels from multiple broadcasters. It obtains channel data from broadcasters and presents it to the user (the user could be human or an application). It allows the user to manipulate, control, and customize the user profile, the received information, and the channels. Depending on a channel's defaults and the user's settings, the receiver is responsible for updating (received/requested) channel content, expiring channel data, and freeing disk space on demand.
Finding of channels can be implemented in several ways. The receiver can query a channel directory or a specialized directory channel that is automatically subscribed. Besides standard channels, there can be specialized maintenance channels that fulfill functions such as maintaining and updating the push system's software components themselves, such as the receiver. Updates of the receiver software are pushed automatically as soon as new versions become available and are installed on the basis of the user profile. Thus users would only have to do an initial setup and could use new software versions immediately. This possibility relates push systems to configuration management approaches like [62] and [63].
Push systems are also related to mobile code systems since channels can distribute executable code. In analogy to applets the notion of a *pushlet* is introduced: executable code and data which is intended for execution at the receiver. Pushlets should execute inside a user-configurable environment provided by the receiver. This imposes new requirements for the receiver. Code must be authenticated (code signatures), the receiving system must be protected from malfunctioning or malicious code that could endanger the receiving system's integrity, and the receiver must supply a user-configurable authorization scheme that can be tailored to the user's requirements and a specific pushlet's needs. Problems and techniques in this area are currently being investigated [55, 108].

2.2.4 Transport System

So far we have completely disregarded the transport system. In a large-scale setting, however, a dedicated transport system is necessary to make a push system scalable and operational, i.e. resulting in decreased network bandwidth consumption and increased availability and responsiveness.

A key design issue for the transport system is access transparency and scaling transparency towards the components and connectors described in the previous sections. Transparency, however, can only be achieved to a certain extent. Both the broadcaster and the receiver need certain knowledge of the transport system to be able to use it. For example, they must know about the concept of repeaters, so that the broadcaster can feed them and the client can connect to a "close" one. This influence, however, should be minimized.
The main protocols in a push system can be derived from the previous sections: subscription protocol, channel protocol (differential updates), backchannel protocol. The transport system should be as transparent as possible towards these connectors.
The components of the transport system can be modeled by a so-called *base distribution component* (BDC). A BDC is a generic component that acts as a broadcaster towards receivers and as a receiver towards broadcasters. This includes understanding the protocols concerned.
A BDC can exist in several configurations:
Repeater. A repeater is preloaded with the channels' contents and offers the same data as the broadcaster but is "closer" (in terms of network properties or some other metrics) to the receiver.
Cache. A cache is like a repeater which is loaded dynamically rather than being preloaded (on-demand repeater). A cache loads data only after it has been requested by one of its clients.
Proxy. A proxy facilitates access to channels where broadcasters and receivers cannot communicate directly, e.g., receivers may be located behind a firewall. Every proxy has a domain translator sub-component that translates back and forth between the generic proxy functionality and the application domain functionality. In the case of a firewall, for example, it translates between the firewall requirements and the push system requirements.
BDCs are organized according to some pattern inside the transport system, for example in a hierarchical structure like a tree. There may be specialized protocols for data exchange between the BDCs to further improve scalability and responsiveness. These protocols depend on the concrete architecture of the transport system and are invisible outside.

2.3 Requirements for widespread Use

The push paradigm has been presented as a programming or architectural model for distributed systems and applications. Such a treatment identifies a number of interesting and important issues for further investigation, including: system-level design and integration issues, business-oriented issues, and multidisciplinary issues that reach beyond computer science and software engineering.
This section presents the main issues that should be addressed in order for push systems to become usable on a wide scale. These issues are also relevant to any architectural model to be used on an Internet scale.
The essential underlying design goal for any distributed system is *scalability*. One of the key issues for a push system is how to manage a large number of subscribers and satisfy their requirements in terms of freshness of information, customization, tailoring, etc. The transport system must minimize resource consumption and network traffic, and distribute computational load. A key requirement for achieving this is cooperation among the transport system components and

between the different organizations running them. As an example of such an organizational structure, repeaters and caches could be collocated with Internet service providers.

The component model of Section 2.2 supports scalability by clearly separating producers from receivers by an intermediate transport system. A standard structure for the transport system based on caching and replication was shown. As will be shown in Section 3.3, the model can be used to accurately describe the structure of existing push systems and to analyze and compare the important design decisions in those systems. But analyzing the scalability of a push system is far from straightforward because it depends on many criteria and many design goals: number of broadcasters, receivers, channels; amount of data on channels; frequency of updates; network latency and bandwidth; and the amount of common subscriptions to certain channels. The component model of Section 2.2 can be used as a basis for developing a scalability model and reference implementations for push systems.

The goal of the Minstrel project (see Chapters 4 through 6) is to develop such a reference implementation using the component model as an architecture for developing plug-compatible components for push systems. Preliminary analysis of the benefits are promising. For example, the worst case for sending a 3.5kB message to 10,000 receivers over a typical mixed-bandwidth network is around 41 seconds, while with standard, non-optimized email this would take over 1 hour. The average delay would be around 13 seconds (see Section 4.6). These figures only take into account bandwidth delays, since the processing load that contributes to the delay can only be estimated (and for Minstrel would be distributed within the transport infrastructure). They still provide a good indication towards performance figures, however, because bandwidth is currently the most limiting resource.

Another key performance issue at the design level involves the choice of locations for repeaters. Sometimes one has no control over this choice, but in many cases it can be influenced. For example, on private (intra-) networks the network provider has complete control over the choice. More interestingly, on the Internet, using Internet service provider sites as repeaters seems to be a promising choice. But this issue, as in many other Internet-related systems, raises the question of payment for services.

Indeed, *payment methods and business models* have to be addressed by any commercial Internet system. This implies that push systems must be able to integrate supporting payment models. Because of the existence of the subscription phase, standard solutions such as macro-payments or flat fee systems (e.g., monthly charge to credit card) may be used. But just as push systems completely reverse the pull model, they also change the traditional payment assumptions. The sender may be interested in charging for all the data it sends out, especially since the receiver has subscribed to the information explicitly, but the receiver is only interested in paying for what is actually read. A standard push architecture supports the investigation of different payment schemes such as micro-payments and pay-per-use. Minstrel includes a generic payment model and standard components for payment schemes (see Section 6.2). Figure 2.5 gives the model for a pay-per-view interaction in Minstrel.

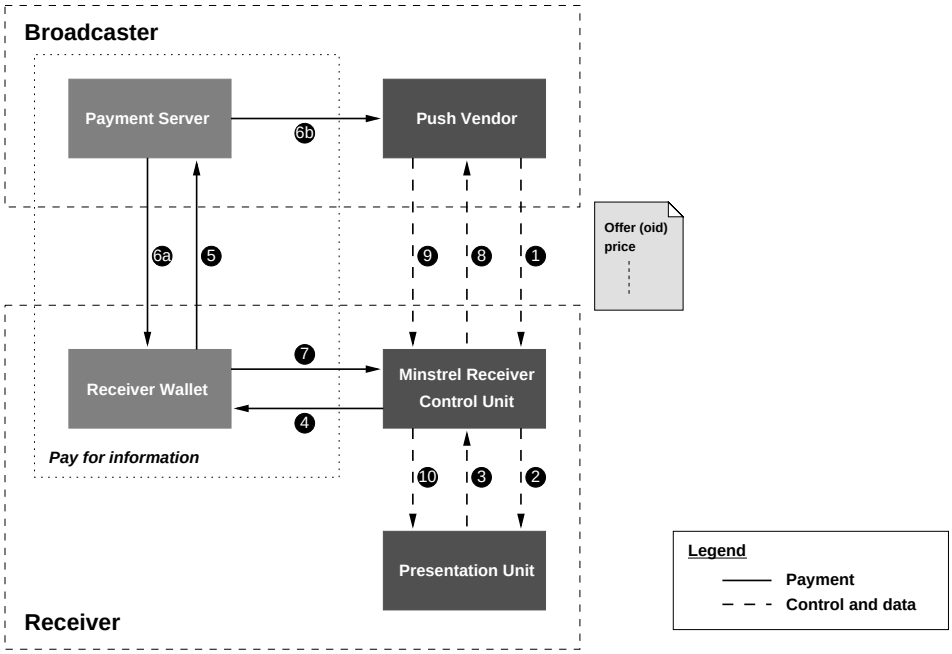

Figure 2.5: Pay-per-view in Minstrel

Say the push vendor has offered some information the user is willing to pay for (1–2). Then the following steps are taken: the user issues a request for the offered information which includes a payment handle, for example, the unique ID of the offer (3). This handle is given to the user's wallet which is instructed to pay (4–5). If the payment succeeds, the payment server sends a receipt to the wallet which in turn notifies the component that processes the user's request (6a, 7). Concurrently, the push vendor is notified and registers the receipt (6b). Now the original user request together with the receipt is sent to the push vendor which checks the receipt and returns the requested data (8–9). Finally, the received data is presented to the user (10). This payment model, which is composed around the notion of a receipt, can also be applied for the implementation of other payment schemes, including time-based or flat fee schemes. A detailed description of Minstrel's support for payment is given in Section 6.2.

Another business-related issue is *security and authentication*. If high-quality information pro-vi-ders want to charge users that receive data via a push system, users must be sure that the information they get is authentic, i.e. fresh, unmodified, and from an identifiable source. This issue is important in typical push application domains like news agencies, financial information services, and other businesses for whom reliability of data is paramount. Technically, it requires the availability of authentication frameworks and certificate authorities on a large scale (X.500, LDAP). For confidentiality of the data itself, the push systems must support encryption methods. Full integration into push systems and "chain of trust" infrastructures—for example, all sites running repeaters must also be trustworthy—still await Internet-scale deployment. Pushlets raise another security problem: How can executable content received from network sources be executed in a safe yet "useful" way, i.e., what accesses to local resources are allowed? The

receiver must provide a flexible security architecture to protect the client from malicious code, i.e. prohibiting intrusion, eavesdropping, or other damages.
As event-based systems, push systems can also facilitate software release management [168], and software deployment and configuration management [62, 63]. Deployment and maintenance of software raises similar security issues as it adds another magnitude of difficulty to the security problems that must be considered by a push system. These problems are similar to the ones that must be addressed by mobile code systems [83].
An interesting research issue involves the application of *software configuration management* techniques to *information configuration management*. A problem that concerns the business community in moving from traditional publishing businesses to push-based publishing on the world-wide web is the packaging and versioning of information. When data may be changed dynamically by sporadic updates, the receiver needs to be able to refer to specific versions of the "documents." With documents that are published at discrete time intervals, such as newspapers, the version numbers can be easily assigned. But versioning of live documents poses many open problems that are currently being addressed in business-oriented research.
For widespread use of push on the Internet, *standard protocols* will be necessary. In particular, protocols and interfaces for channel definition, subscription, and access will be needed. At the moment, the available push systems are incompatible and cannot interact. Thus users and information providers have to install dedicated software for each system, and information needs to be tailored and structured explicitly for every system supported. A unified framework/standard as exists for the Web is necessary to make push systems a successful technology.
Moreover, a *quality of service* (QoS) concept for push systems is needed to provide qualitative assessment of such systems. This would support qualitative comparison of push systems and allow a user to choose a push system that fits his/her requirements. The model defined in this chapter provides a basis for comparing push systems structurally—as will be done in Section 3.3—but does not define the QoS. However, it provides a good starting point for a QoS model, since a wide range of QoS properties can directly be inferred from it.

2.4 Summary of the Model

Even though there are many documents on the world-wide web and in electronic magazines about push systems, these are mostly at the user and application level, with little systematic treatment of the design and research issues. This chapter has presented push systems as an architectural model for distributed systems and interactions and has positioned it with respect to client-server and event-based architectures. The subscription phase of the interaction model is the key to the scalability of the push model and is applicable to many distributed applications for which client-server computing is deficient. This chapter has presented a communication and component model for push systems that may be used to study, analyze, and contrast different implementations of push systems (which will be done for six prominent push systems in the next chapter). Using the concepts of information source, receiver, broadcaster, and transport system, the component model separates the issues of content management, channel management, scalability, and user-interface management into different components. The component model may also be used as a

basis for a reference implementation of push systems. Moreover, the main issues were presented that need to be tackled by push systems: scalability, network traffic, security, authentication, and electronic commerce. The Minstrel system described in the second part of this thesis addresses all of these issues and is a proof-of-concept implementation of the architectural model.

Chapter 3

Related Work

The concept of actively disseminating information is not new and not restricted to the push area. On the contrary, it has always been one of the premier goals of distributed computing systems. Since the early days of the Internet, services like electronic mail and Usenet news have been used for this purpose. With the development of the Internet and its underlying network infrastructure, many additional approaches for this application domain have been devised. The most prominent and recent of these that are related to push systems are multicast infrastructures such as the MBone [40], event-based systems, and mobile code systems.
This section presents concepts and approaches related to push systems and in particular to the Minstrel approach described in Chapters 4 through 6. It starts with a brief description of existing multicast infrastructures and protocols that can be exploited by push systems or have been devised as push-like systems themselves in Section 3.1. Section 3.2 then considers electronic mail and Usenet news as still valid and widely used alternatives to the push approach. In Section 3.3 six prominent push systems are presented and classified on the basis of the communication and component model of Chapter 2. Since a great variety of push systems exists, this comparison has been restricted to the most relevant ones. Section 3.4 briefly overviews other interesting push systems. Closely related to push systems are event-based systems, which received much attention in the research community recently. Section 3.5 compares event-based systems with push systems and overviews event-based systems to define the design space of such systems. The final section briefly addresses the relations between push systems and mobile code systems.

3.1 Multicast Infrastructures and Protocols

Multicast infrastructures aim at a communication pattern similar to push systems. In contrast to standard networking information infrastructures that exhibit a 1:1 relation between sender and receiver, they target a 1:n or even m:n (bi-directional) relation between sender(s) and receivers. The availability of such communication primitives at the networking level greatly simplifies the dissemination problem at the application level (e.g, for a push system). Push systems can exploit multicast infrastructures to notify consumers of the availability of new data or distribute the data itself via multicast. Several approaches which exploit multicast infrastructures, for exam-

ple, WebCanal (see Section 3.3.5) or Keryx (see Section 3.5.1.2), will be described later in this chapter.
Multicast simplifies the architecture and implementation of push systems and has several efficiency benefits. Many of the issues which must be addressed by the broadcasting process and the transport system can be delegated to the networking level if multicasting is available. Multicast protocols for the Internet scale reasonably well and if used with a push system may even make the push system's transport system obsolete. Many multicast protocols include real-time facilities that can be exploited to guarantee timeliness of data and allow for streaming multi-media contents of channels. At present, however, only a limited number of users have access to such resources.

3.1.1 MBone

The multicast backbone (MBone) [40, 87, 99] is a world-wide virtual network that implements multicasting. Multicasting means that one host sends to a group of hosts. The following description is based mainly on [40] and includes some excerpts that are not explicitly marked.
Multicast groups are modeled by special IP addresses: On the Internet, the IPv4 address range from 224.0.0.0 to 239.255.255.255 is reserved for multicast addresses. When a host wishes to join a *multicast group* represented by one of the IP addresses in this range, it issues an *Internet Group Management Protocol* (IGMP) request. If a host joins a multicast group, this means that it wishes to receive from and/or send to that group. A specialized *multicast router* which is responsible for the host's subnet will then inform other multicast routers so that multicast packets are sent to and received from the host's subnet.
This can be viewed as a "subscription phase" for multicast packets. However, this notion of subscription is different from a subscription in a push system. With multicasts over MBone, the sender does not know who will receive its packets. The sender simply sends to an address, and it is up to the receivers to join that group (i.e., the multicast address).
MBone is composed of networks (called *islands*) that support multicast [40]. An example could be Ethernet LANs that support multicast because they use broadcast packet distribution, which also comprises multicast. Each island has a dedicated host that runs the multicast routing daemon *mrouted*. These daemons that virtually represent islands are connected with one another via unicast *tunnels* (1:1 relation). Figure 3.1 depicts this configuration.
Each island consists of a number of *client hosts* (C) that are connected via a local network and one *gateway host* (M1, M2, M3) running *mrouted*. The gateway hosts are connected via point-to-point tunnels. Sending a packet via this infrastructure works as follows. A client sends a packet to a multicast address. Via the local network this packet is delivered to the gateway host that is responsible for this subnet. *mrouted* will consult its routing tables and decide over which tunnel(s) to send this packet in order to reach all members of the multicast group. The *mrouted* at the receiving end of a tunnel will check whether clients on its subnet are subscribed to the packet's multicast group (multicast address) and if clients are indeed subscribed, *mrouted* will forward the packet to the local subnet. It also decides whether the packet has to be forwarded to any other tunnels.

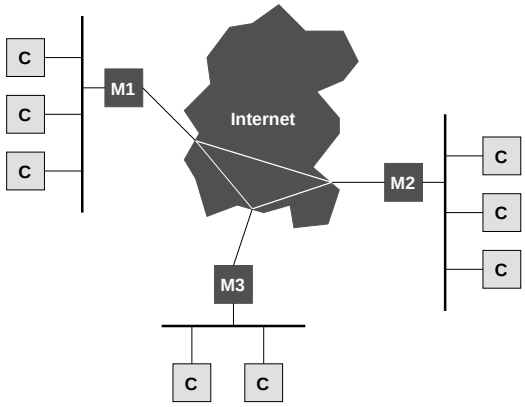

Figure 3.1: MBone architecture (islands and tunnels) [40]

To summarize, MBone provides multicasting by connecting multicast subnets via point-to-point connections and offering a routing facility that handles the distribution of packets to all receivers. MBone's islands are not managed centrally but coordinated via mailing lists. All traffic in MBone uses the User Datagram Protocol (UDP), though most MBone applications are based on the Real-Time Transport Protocol (RTP) [150] which is built on top of UDP. These protocols mainly target streaming data like audio and video, where occasional packet losses can be compensated. For the transmission of data that requires accuracy, such as web pages or program code, one needs additional protocols such as LRMP (see Section 3.3.5) that provide reliable and ordered packet delivery services.

3.1.2 Systems based on Multicast

Many tools exploiting MBone's functionality exist, including audio and video conferencing tools, real-time audio and video, whiteboards, group messaging tools, image multicasting facilities, and push-like applications. Among these are *liveCaster* [98], a tool to multicast MP3 audio but also short textual messages (receiver software is available, too), and *multikit* [100], a distributed multicast directory browser, that can be used for announcements. *multikit* provides an interactive program guide for Internet multicast services that also allows the user to create own announcements and organize them.
Several systems similar to push systems use MBone for real-time distribution of HTML pages. *webcast* [117] enables a group of Mosaic web browsers [119] to share a set of web documents via the MBone. It exploits Mosaic's Common Client Interface (CCI) [118] and uses the Reliable Multicast Protocol (RMP) [157, 177] for distribution.
mMosaic [29] is a tool for sharing web pages over the MBone. It allows transmission and interactive use of web pages based on a version of XMosaic that is extended with multicasting functionality. This extended version is used as both receiver and sender.
mWeb [133] is a framework for distributed real-time web presentations via the MBone: First, a set of web pages is multicast to clients via the Scalable Reliable File Distribution Protocol

(SRFDP); then, when the presenter steps through this list of pages, all clients are synchronized via the WebDesk Control Bus to display the relevant page that is already available at the client's site.
MultiCast Mosaic [165] is a tool for multicasting HTML slides over the MBone. It is based on a client-server model: A master repeatedly multicasts a package of slides to its clients in the hope that at least one error-free copy is received by all remote sites. Later, when the master loads one of the pages in the package, the relevant URL is multicast to the clients. On receipt of this notification, the clients display the corresponding page from their received package.
An interesting approach is suggested in [143]. It takes a different view of the notion of push and suggests that popular and frequently changing web documents should be distributed using continuous multicast push (CMP). The authors argue that popular web sites create "hot spots" on the Internet where the same data is transmitted over the same links again and again. While for rather static data this problem can be remedied with caches, this is not possible for frequently changing data. Therefore, short-lived information such as stock market data should be delivered directly to consumers using CMP.
Several push systems directly or indirectly rely on multicasting facilities, since it alleviates the requirements the push system has to fulfill in the broadcasting process. One such push system is WebCanal [95, 96], which is described in Section 3.3.5.

3.1.3 Real-Time Transport Protocol

The Real-time Transport Protocol (RTP) [150] provides end-to-end transport services for transferring real-time data over multicast or unicast networks. If the underlying network supports multicast, RTP can exploit this functionality to provide data transmission to multiple destinations. RTP is independent of the underlying network. It typically runs on top of UDP and exploits UDP's multiplexing and checksum services, but can be used on any packet-switching network such as ISDN or ATM networks. Many MBone applications use RTP and it is also used indirectly by many push systems and event notification services that are based on the MBone. RTP mainly targets streaming data like (interactive) audio and video where occasional packet losses can be compensated.
RTP's services include payload identification, sequence numbering, timestamping, and delivery monitoring. It does not ensure timely delivery and does not provide quality-of-service-guarantees. RTP provides "best effort" transmission where packets can be lost without retransmission and packets can be delivered out of sequence. These drawbacks, however, can be compensated by higher-level protocols that exploit RTP's packet sequence numbering and other services.
The RTP RFC [150] actually defines two closely related protocols: RTP to carry the payload information and the RTP Control Protocol (RTCP) that conveys meta-information about an ongoing session, such as information about the participants and how well they receive the data. On the basis of this information, senders can be requested to adapt their transmission rates to the current packet loss.
Because of its best-effort approach, RTP cannot be used directly for push or event-based systems. For transmission of data that requires accuracy, such as web pages, program code or events,

higher-level protocols are necessary that provide reliable and ordered packet delivery services. However, RTP can be used as a substrate for such higher-level protocols.

3.1.4 Reliable Multicast Protocols

For push and event-based systems, reliability is a key issue. Unlike with continuous data, ordered and reliable delivery matters. Packet losses cannot be compensated. This section presents some reliable multicast protocol approaches that have been devised for push systems and similar services or can be exploited for this purpose.

3.1.4.1 Scalable Reliable Multicast

Scalable Reliable Multicast [45] is a reliable multicast framework for application-level framing and light-weight sessions. SRM is robust and efficient and scales well to large networks and large sessions. The framework offers packet loss detection and repair and can thus be used for applications that require a reliable data delivery service.
SRM relies on three main concepts to provide a reliable multicast service: *heartbeats*, *negative acknowledgments* (NACKs), and *repairs*. Each member periodically sends out a heartbeat with the sequence number of the latest packet. This information can be used to detect packet loss. If a packet loss is detected, a NACK packet is sent to report the packet loss to all participants. If a participant receives a NACK, it sends the lost packet to the group as a repair. To accomplish this each participant caches the latest packets. To avoid NACK and repair "explosions," the algorithm uses random timers at every participant.
SRM has been prototyped in a distributed whiteboard application. It scales well and has been extensively tested on a global scale with sessions of up to 1,000 participants.

3.1.4.2 SRRTP

The Scalable Reliable Real-time Transport Protocol [132] (SRRTP) incorporates parts of the SRM framework into RTP by extending both the RTP and RTCP protocols. Its intention is to provide a scalable and reliable multicast data transmission facility for transporting a data flow reliably over the transport protocols supported by RTP.
SRRTP is based on packets. To support distribution of higher-level aggregations of packets such as files, which is the typical granularity for distributing information via push-like systems, an additional protocol, the Scalable Reliable File Distribution Protocol (SRFDP) is proposed. SFRDP exploits SRRTP and can reliably send files, groups of files, and meta-information about these files.
SRRTP and SRFDP have been deployed in the mWeb application [133] described in Section 3.1.2.

3.1.4.3 Reliable Multicast Protocol

The Reliable Multicast Protocol [157, 177] (RMP) provides a totally ordered, reliable, atomic multicast service on top of an unreliable multicast service such as IP multicasting. It offers a

wide range of delivery guarantees, including agreed and safe delivery, that are scalable on a per packet basis. Both a publisher/subscriber and a client/server model are supported for message delivery.
RMP provides an implicit naming service that maps textual group names into communication groups. To increase scalability it allows processes that are not members of a group to send messages to and receive messages from the multicast group (multi-RPC mechanism). RMP is based on a token ring technique which makes it not well-suited for wide area network applications. It is used in the *webcast* system [117] described in Section 3.1.2.

3.1.4.4 Light-weight Reliable Multicast Protocol

The Light-weight Reliable Multicast Protocol [94] (LRMP) is a general purpose reliable transport protocol over unreliable underlying network protocols. It is based on RTP and a modified version of SRM. The main features it offers are loss repair, ordered packet delivery, and adapted rate-based flow control.
To provide better scalability, the loss-repair algorithm of LRMP uses local loss repair: Only the sender is allowed to send repair packets. This contrasts with SRM, where all participants are involved in the repair process. To provide in-order packet delivery, out-of-order packets are retained in the cache until the correct sequence has been re-established (repair packets). The flow control feature of LRMP allows it to keep an acceptable speed for the majority of receivers and tries to satisfy slower receivers when possible. The transmission rate is dynamically adapted to suit the available network bandwidth.
LRMP is deployed in the WebCanal [95, 96] system that is described in Section 3.3.5.

3.1.4.5 Multicast File Transfer Protocol

StarBurst's Multicast File Transfer Protocol [113] (MFTP) is intended to provide an efficient and reliable transfer facility for data organized as files from a sender to multiple receivers. It has been optimized for file delivery rather than providing a generalized multicast transport layer and is not intended for real-time or streaming data. This goal makes it well-suited for the application domain of push systems, where the typical distribution units are files rather than packets. Additional design goals are simplicity to increase reliability and scalability to high numbers of users.
MFTP is based on UDP and consists of two parts: an administrative protocol for group and session maintenance (Multicast Control Protocol) and a data transfer protocol (Multicast Data Protocol) that is in charge of sending files simultaneously to members of a group. MFTP can operate with networks that support broadcast or multicast at the data link layer such as LANs and multicast IP networks. If multicasting is not available, "application layer multicast" is employed. This means that broadcasting is used where available (e.g., in a LAN) and sequential unicast where it is not (after an MFTP host has applied filters so that messages get sent only to the members of a defined group).
The sender (server) in an MFTP setting continuously sends data without waiting for responses from receivers (clients). Clients only send responses for data transmission units (DTUs) that they

did not receive. MFTP supports checkpoint/restart functionality so that an interrupted transfer can be resumed without having to retransmit the previous data. A maximum server transfer rate can be specified that allows limitation of the bandwidth used by MFTP, so that other applications can use the remaining bandwidth.
MFTP is deployed in StarBurst's OmniCast content distribution application and is used by several push systems, such as BackWeb (Section 3.3.2) and Microsoft Webcasting (Section 3.3.3), as a multicast transport medium. .

3.2 Alternative Approaches

With a relaxed definition that focuses on the underlying publish/subscribe model, there are several systems the can actually be viewed as "push systems." This section briefly contrasts such systems with true push systems.

3.2.1 Electronic Mail

The primary functionality of push systems is information dissemination. Electronic mail [24] was one of the first services on the Internet that was deployed on a large scale for this purpose. It is still one of the basic and most important services of the Internet. Initially, email was intended for the transmission of limited-length text messages to one or a few recipients. In the meantime, email has been enhanced in many respects, so that it can now be viewed as an approach worth comparing with push systems.
The introduction of mailing lists, which was a simple improvement to the original design, provided a powerful tool for establishing bi-directional, one-to-many relations. Efficient tools for creating and automating the maintenance of mailing lists, such as Majordomo [19], have since become available. These tools typically facilitate the subscription to and unsubscription from mailing lists which previously had to be done manually by the list administrator (Some moderated mailing lists are still managed in this way).
Majordomo, as a representative example of such tools, also provides further administrative functions to minimize human intervention in the maintenance of a mailing list. It allows users to perform some typical administrative operations by sending mail with the according commands to a special, typically list-specific, mail address. Besides subscription and unsubscription requests, these functions include automatic responses to common user requests such as the topic of a certain mailing list, a "directory" of mailing lists served by a specific Majordomo site, information on the subscriptions a user already has, or simply automatic help messages which explain the commands and functionalities of Majordomo. Additionally, Majordomo provides functions that help the list administrator (list owner) to monitor his/her mailing list(s) or approve the operations requested by the users (e.g., subscription). This setup provides the typical publish/subscribe interaction pattern of a push system.
As noted above, mail was originally intended for text messages of limited length (e.g., 1,000 characters or less as specified in [137]). This forced users to convert non-textual content into a 7bit US-ASCII text representation and split the content over several mails. Such drawbacks were

overcome with the introduction of the Multipurpose Internet Mail Extensions [46, 47, 48, 49, 114] (MIME). MIME mails can transport arbitrary content of (conceptually) arbitrary size.
MIME mail can be exploited to make electronic mail look even more similar to a push system. Via appropriate MIME types, executable content can be transported to a receiver, unpacked there, and executed at the receiver's site. Several approaches have been devised to accomplish this, for example D'Agent [57] (formerly known as AgentTcl).
Email can also provide authentication and security as available in many push systems. Based on the MIME standard, the S/MIME standard [31, 37, 38, 73, 74, 139, 140] defines MIME document types and a set of security services to provide message authentication, message integrity, and non-repudiation (using digital signatures) and privacy and data security (using asymmetric encryption).
For authentication purposes mails can be signed: The sender runs a cryptographic hash function over the content to be signed, encrypts the result using his/her private key and attaches the resulting signature to the mail. The receiver can then run the same hash function over the received content and compare it with the attached signature that s/he first decrypted using the sender's public key. To get the sender's public key, the receiver queries a repository (e.g., a web site), that provides public keys. To provide privacy and data security the content of the message can also be encrypted (using an asymmetric encryption method). This framework can be used not only with mail but also with other systems. It works with traditional mail user agents, provided that both the sender and the receiver have an adequate cryptography tool, such as PGP [52].
The capabilities of email can be summarized as follows: Email in conjunction with appropriate tools provides a 1:n bi-directional publish/subscribe infrastructure that is available on an Internet scale, can transport arbitrary data, and has authentication and security services. Users can select topics they are interested in by choosing and subscribing to an appropriate mailing list. Mailing lists can be viewed as "channels". Additional filtering can be applied with mail processing tools like *procmail* [167]. This raises the question why people devised push systems instead of simply using email—an existing technology already implemented on a global scale. The answer to this question is not simple.
Most importantly, it is a matter of resources. While mailing lists facilitate the sending of mail to any number of users, this can have serious impacts on resource consumption if the number of subscribers is high. The typical mail distribution method [21] duplicates every single mail sent to a mailing list for every receiver on the list. Suppose a 300kB image file is to be sent to 1,000 receivers. Then this mail is duplicated 1,000 times and the resulting mails use up 300MB of disk space on the sender's machine. This is the figure for one (!) mail. It may get even worse if the mail frequency is high enough. Since the delivery of mails depends on the reachability of the receivers, figures may add up rapidly. Additionally the processing load and network bandwidth consumption for delivering the mails is placed solely on the sender's machine and network connection. As 300MB of disk space are required, 300MB of network bandwidth will be needed to deliver all mails in the previous example. It may even be that a mail is delivered multiple times to the same mail relay host if this host is in charge of handling the mail exchange for several receivers. This shows another drawback of mail: Mail transport agents such as sendmail [21] communicate directly without an intermediate distribution infrastructure, which means that the mail server of the sender communicates directly with the mail server of

the receiver. Figures provided in [86] for a 2,000-subscriber mailing list and 200 messages a day show a total of 400,000 messages a day, which causes considerable computing load and delivery delays of up to 5 days.

To overcome these problems, several strategies have been suggested. [86] proposes complex configuration, tuning, and load balancing strategies with a high degree of parallelism—both in terms of parallel sendmail processes and distribution of the queuing load over several machines—and splitting large mailing lists into smaller ones based on delivery delays. This can cut the time to deliver 95% of the mail queue from 5 hours to 3.5 minutes for the mailing list mentioned in the example above. This solution was developed only after thorough analysis with newly developed tools.

Similar techniques are suggested in [18]. Many operating system configuration and tuning measures were applied in this approach using newly developed analysis tools. Specialized load-balancing and mail queue post-processing software had to be developed. It is interesting to note that MX record optimization did not prove very helpful. MX record optimization means that receivers are sorted according to their mail relay host. Thus only one copy of the mail is sent to a relay host, with all the receivers serviced by this host listed in the address field. Unfortunately it turned out that out of 5,000 individual subscription addresses, 4,000 were unrelated by host, domain, or MX record.

Both [18] and [86] also include comprehensive analyses of the problems of large mailing lists. However, it is unclear whether the suggested techniques can be generalized and deployed on a large scale and be applied to other operating systems.

Additional problems are introduced by mail bounces and “spam” mails. A mail bounce means that a mail was not deliverable and has “bounced back” to the sender. This may cause additional loads on the mailing system. “Spam” mail means unsolicited mails that are sent to a mailing list. To prevent such mails, sender addresses have to be checked before a mail is accepted for delivery, which is not typically done by default. Even this precaution is not totally effective, since mail headers are very easy to forge.

Forging mail headers connects to the issue of authenticity. While content can be signed and secured, mail headers are not protected in any way. A mail message that has been received and contains authentic data can easily be redistributed by anyone else. This is especially a problem if data (e.g., stock data) is replayed, at a later time and with a forged sender’s identity (e.g., with the identity of the original sender).

Also the mail transport system needs very proper configuration to be immune to attacks like denial of service. Without a properly configured sendmail mail transfer agent, which, for example, restricts relaying, “spammers” could use an unprotected mail relay host to distribute their mails at the expense of the mail host’s owner.

A fully featured mail system as described above that would closely resemble a push system depends on many individual components that all need to be configured very carefully. These components must also work together properly to provide all needed functionality. Due to the loosely coupled architecture that lacks integration of the components, this is fairly difficult to achieve and may open severe security holes if not done with great care.

A functionality that is currently missing completely from electronic mail is support for payment and e-commerce. So far no technology or system has been devised that would support payment

by means of email. It is difficult to envision such a system in any case, because of the problem of loose coupling mentioned above.
To summarize, electronic mail has most of the characteristics necessary for the implementation of a push system on top of it but has severe drawbacks due to its decoupled architecture.

3.2.2 Usenet News

Usenet news [90] offers a worldwide distributed blackboard on top of other networks. It is divided into hierarchical discussion forums, called *newsgroups*, which are dedicated to defined topics. Newsgroup topics span a wide range, from recreational activities to scientific issues or computer-related topics. A rudimentary navigation facility among the available newsgroups is provided by the hierarchical dot notation of newsgroups. For example, the *comp* newsgroup hierarchy deals with computer related topics, *comp.lang* narrows the focus of newsgroups to programming languages, and the specific newsgroup *comp.lang.java* focuses on the Java programming language.
Users can access Usenet news, commonly denoted as news, via a news reader (client) which provides the user interface and manages the interaction with the news server. Users can *subscribe* to a set of newsgroups based on their interests. After selecting a specific newsgroup, the user can read contributions (*articles*, *postings*). The user may submit replies to articles or submit (*post*) new ones. The news infrastructure then takes care of the worldwide distribution of the postings. For focused discussions inside a newsgroup, postings can refer to each other by using article identifiers provided by the news system. News reader programs exploit these relations for conveniently grouping the articles when presenting the contents of a newsgroup to the user (*threading*). To reach a wider audience, articles can also be posted to a set of newsgroups (*cross-posting*) which, however, should be used economically to avoid additional traffic and keep discussions focused to one newsgroup. For focusing replies to an article to a dedicated newsgroup, a so-called *follow-up* newsgroup can be specified. This means, for example, that a user can post an article to 20 newsgroups and define a follow-up in the posting that news reader programs would recognize and direct replies to that newsgroup only. Currently about 55,000 newsgroups are available and news traffic and the number of newsgroups are growing steadily.
Inside the news system, news articles are distributed using the Network News Transfer Protocol [82] (NNTP). Figure 3.2 shows a simplified news network.
The news network consists of two classes of software: news readers (news clients in Figure 3.2) and news servers. News readers provide the user interface to the news infrastructure, with which the user can subscribe to newsgroups, read and post articles, etc. News reader programs interact with the news infrastructure (consisting of news servers) via the Network News Reader Protocol[1] (NNRP). Each news reader communicates with exactly one news server. Larger organizations such as companies, universities, or Internet service providers typically run one centralized news server for their users.

[1] NNRP is actually a subset of NNTP which is used by news clients. It is commonly referred to as NNRP and is not a protocol of its own.

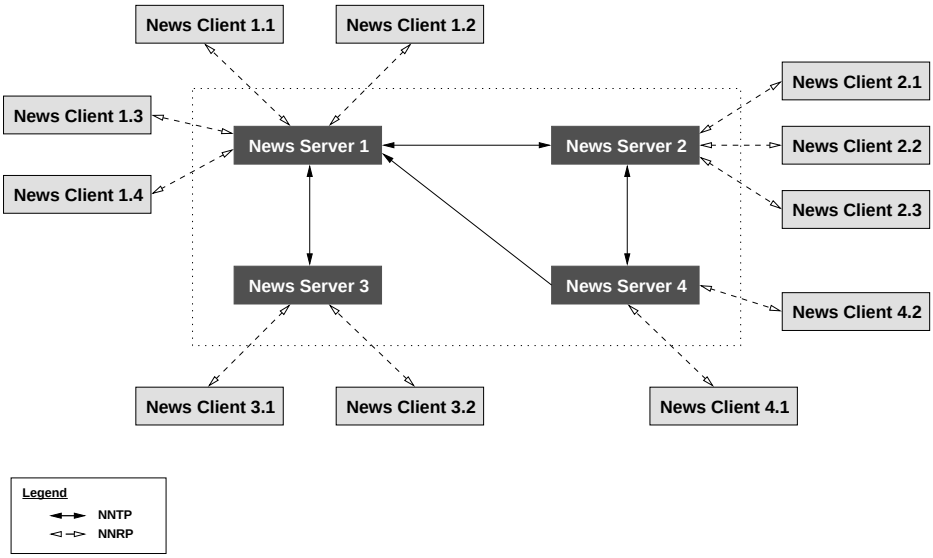

Figure 3.2: A simplified news network [59]

The worldwide virtual network (the so-called Usenet) of cooperating news servers is the distribution infrastructure of the news system. The Network News Transfer Protocol [82] (NNTP) is used to propagate articles among news servers. News servers receive articles from their clients and other news servers. Propagation of articles is commonly referred to as *news feeding*. Each news server defines which newsgroups it wants to hold, i.e. which newsgroups its clients can access, and has a statically configured set of servers that constitute its news feed.
An article posted by a client is transferred via NNRP to its dedicated news server, which is the client's access point to the news infrastructure. Each news server then keeps a copy of this article and propagates it to its neighboring news servers. This propagation finally results in a copy of each article on each news server in the infrastructure. Since no restriction exists on the topology and distribution paths of the news system, servers may receive an article multiple times. However, each article carries a globally unique identifier that allows the news servers to identify such duplicates. Causal ordering of articles is not provided: The different distribution paths of articles may cause that a reply reaches a news server before the article it refers to.
Through the setup described above, Usenet news exhibits several properties of a push system. It offers topic-based subscriptions to newsgroups, which models the concept of channels in some respect; it has an n:m relationship model that supports bi-directional communication of data to a large number of users; and it is deployed and available on a world-wide scale. However, news has several drawbacks, if it is considered as a push system.
The biggest problem of Usenet news is that it is no longer scalable without substantial redesigns [59]. The increased amount of data that needs to be transferred by the news infrastructure causes severe problems. An enormous redundancy is caused by copying each article of a newsgroup to all the news servers holding this newsgroup, with only a small fraction of the articles actually being read. A full newsfeed currently requires ˜400kBit/sec of bandwidth (2.5GB/day).

This means that 27% of the bandwidth of a T1 link (1.5 MBit/s), which is the typical Internet link for many sites, is dedicated to the news service. A further description of the infrastructural problems of news and possible solutions are given in [59] and [60].
The consideration of other properties of push systems, such as security and authentication, renders news a problematic candidate. Although news readers could be enhanced with authentication and security services based on the S/MIME standard [31, 37, 38, 73, 74, 139, 140], this is not being deployed. As with mail, forging the origin of a posting is simple since NNTP headers are not protected and are not verified by the news system. Thus, if a trustworthy information flow is required (stock data, news agencies, etc.), news cannot be used.
As electronic mail, Usenet news has no integrated concept for payment and e-commerce. So far, no technology or system has been devised that would support payment for news. It is hard to envision such a system in any case because of the decoupled design of news. The statements in Section 3.2.1 concerning loose coupling also apply to news. Unlike email, no approaches exist for the support of mobile code. However, news has been used for many years as the primary medium for distributing free software.
No restrictions exist on who is allowed to post to a newsgroup (except for a few moderated newsgroups where a human user decides what gets posted to the newsgroup). This opens the door for "spamming."
Moreover, news does not support timely and ordered delivery of data. First, the distribution process itself has no upper limit for the distribution delay, and second, users have to check the newsgroups they are interested in for articles regularly. Additional filtering and customization is not available.
This list of deficiencies is not comprehensive but explains why Usenet news is no viable basis for a push system.

3.2.3 World-wide Web and Mail combined

The combination of the world-wide web and electronic mail provides an interesting push-system-like approach. Such combined services offered by several portal sites such as Netscape Netcenter [126] rely on a hybrid "broadcasting" scheme similar to the hybrid broadcasting approach described in Section 2.2.2.1.
First, the user visits a web site, such as Netscape's In-Box Direct service [122], where he/she can select from a set of "publications" on a variety of topics ("channels"). This selection process actually means the user is subscribed on a number of mailing lists corresponding to his/her selections (subscription phase). Now the user will receive mails from the subscribed channels at regular intervals.
The received mails typically hold an HTML document that is displayed as in a browser when read with an appropriate mail tool such as Netscape Communicator (push part). The HTML document typically looks like the frontpage of a newspaper and holds links the user can click on to retrieve the corresponding documents (pull part).
This approach resembles the sample-shipment-based hybrid broadcasting scheme used in Minstrel (Chapter 4) but is less integrated. However, it addresses several of the problems described in Section 3.2.1: The HTML documents that are distributed are rather small (around 15–20kB)

since they hold only the pure HTML part of the page, while images and other content referred to on the page are loaded when the page is displayed (this, however, requires the user to be online). Thus the displayed page looks like a normal web page, the only difference being that it was not loaded from a web server but delivered directly to the user. Even such relatively small mails, however, can cause scalability problems for the sender's mail system, as is described in [18] and [86].

A major drawback of electronic mail is its lack of support for payment and e-commerce. This problem can be circumvented by this hybrid setting, in which mail only serves as the notification medium. By clicking on a link in the delivered page, the user typically loads the selected web page in a standard web browser window and can use all functionality available from web sites, including support for electronic payment. Most current electronic micro-payment and macro-payment systems, such as Millicent [54] or SET [151, 152, 153, 154], are targeted at web sites. Thus feasible infrastructures are already available and await large-scale deployment with web servers.

Another security related problem described in Section 3.2.1 is alleviated by this setting: Forging the sender's email address has less impact on the system and the user, since the user typically has to go to a web site for further information and can recognize whether received information is authentic. It is also very likely that the companies running such services only allow messages from a very restricted number of sites, which almost completely eliminates the problems of forgery and "spam."

This combined configuration offers several advantages: The underlying technology is accessible to many users; users can specify their interests in a reasonable way; pre-selected, presumably high-quality content is distributed; and the infrastructure can also be offered as a publishing infrastructure to content providers. Such systems are the most relevant competitors of push systems. Despite their weaknesses, such as lack of integration, limited support of mobile code, limited customization and filtering, underlying protocols (SMTP, HTTP) that were not designed to support the push system application domain, and scalability problems of the distribution infrastructure, they are a viable approach and have gained widespread deployment.

3.3 Representative Push Systems

Since 1996, a number of commercial systems have appeared that classify themselves as push systems. In this section six prominent examples of such systems are surveyed. Table 3.1 compares the systems with respect to components and Table 3.2 classifies them in terms of the main features that were described in Chapter 2. Providing such a comparison is surprisingly difficult due to the paucity of technical documentation on these systems. It was not possible to find the answers for some of the entries in the tables. In the following, each system is examined briefly.

Push System	Channel	Broadcaster	Comm. Paradigm	Transport System
Castanet	√	√	pull	repeater, cache, proxy
PointCast	√	CBF	pull & limited push	cache
BackWeb	√	√	pull & push	cache
Webcasting	√	–	pull	–
WebCanal	√	√	push	–
Intermind	√	–	pull	–

Table 3.1: Comparison of push systems based on the component model

Push System	Back-channel	Pushlets	Update Strategy	Filtering	Scalability	Receiver Update	Data Sec.
Castanet	plugin	√	diff. (byte)	–	high	√	high
PointCast	–	limited	?	limited	low-medium	√	–
BackWeb	√	√	diff. (byte)	√	medium-high	–	high
Webcasting	external	√	diff. (file)	–	high	√	low
WebCanal	R = B	browser-like	diff. (file)	–	low-medium	–	–
Intermind	external	browser-like	?	limited	medium-high	–	–

Table 3.2: Comparison of push systems based on features

3.3.1 Castanet

Castanet [105, 106, 107] is an advanced push system for distributing content with specific emphasis on software deployment over the Internet. Software can be downloaded and installed and kept up to date. Obsolete versions are removed automatically. This functionality is based on the Open Software Description Format [170] (OSD), a joint effort of Marimba, the manufacturer of Castanet, and Microsoft. The goal of OSD is to provide an XML-based vocabulary for describing software packages and their dependencies. OSD supports the specification of software packages as a directed graph indicating software dependencies (one package requires another) and thus allows for free modularization of software package descriptions with software packages being the nodes of the graph. This dependency graph is mapped onto an XML [14] description that is sent to a consumer which then can determine what packages it may need.
Castanet supports several types of channels: *file collections*, *HTML*, *presentation*, *applet*, and *application channels* or combinations of these. A file collection is simply a set of files that can hold arbitrary content types and are to be distributed to consumers. An HTML channel is a collection of web pages under a common directory. The other three channels types are so-called executable channels, meaning that they hold executable program code. A presentation channel is a Java application with a graphical user interface and optional scripts based on Marimba's Bongo presentation builder. An applet channel is a Java applet and an application channel comprises various application types such as Java applications and Visual Basic applications.
Castanet's broadcasting paradigm is pull-based: The client pulls at intervals configurable down to 15 minutes to download newly available channel content, or the user may issue a pull request. An update schedule for a channel can be defined by the provider. Updates are differential on a

byte granularity, i.e., only updated parts of data in a channel are sent to the receivers. Updating is done via the Distribution and Replication Protocol [169] (DRP) which has been submitted to the World-wide Web Consortium (W3C). It has been designed to efficiently replicate a hierarchical set of files to a large number of clients. To determine whether a client has an up-to-date view of a channel's content, Castanet uses hash sums. This makes the checking process very fast and supports short online periods for dial-up users.
A limited backchannel functionality is provided by *plugins*: one plugin per channel allows processing of feedback data, e.g., return language-specific data based on the user's configuration. Additional backchannel functionality is available through appropriate functionality that can be provided inside the executable channels. This, however, is outside Castanet's framework. No explicit means of filtering exists, but a limited degree is possible via user configurations.
Castanet's transport system provides for high scalability due to its transport system infrastructure, which has repeaters, called *transmitters* (the broadcaster is called *primary transmitter*), caches (called *proxies*), and proxies (called *gateways*) that allow channel access behind firewalls. When a primary transmitter receives a subscription request, it can automatically assign the receiver to a specific repeater. A strategy for this has to be provided by the maintainer of a channel. The information source is modeled by the *publisher* software component. It supports the creation of channels and provides the necessary functionality to publish a channel via a transmitter.
Transmission of data is efficient because of Castanet's differential update strategy and the fact that multiple modified files are sent over a single network connection. The Castanet receiver (called *tuner*) is a channel itself and can be automatically updated. Castanet supports two security concepts: *Channel signing* guarantees the integrity and authenticity of channel data, and SSL [128] provides encrypted transmission. Channel signing can be exploited to create trusted channels that have special access rights at the receiver's site.
For announcement of channels, specialized channels and other infrastructures exists. Notifications of new data can be done with respect to user profiles and user groups. A version of Castanet is part of the Netscape browser (*Netcaster*).

3.3.2 BackWeb

BackWeb [6, 7] is a highly configurable framework for information distribution. It comes with a rich set of supporting applications and authoring tools, including a specialized authoring language—*BackWeb Authoring Language Interface* [5] (BALI). Like Castanet, BackWeb can be used for software deployment, but it has less powerful concepts and tools than Castanet.
BALI is a scripting language for authoring animated and multimedia *InfoPaks*. Simple "programs" (scripts) can be written in BALI that describe and control the behavior of an InfoPak. Scripts typically define multimedia animations. BALI supports parallel execution by its *sprite* concept. A sprite is a simple form of a "process" (or "thread").
An InfoPak is a logical collection of files that forms the unit of information distribution in BackWeb. An InfoPak consists of the data to be distributed and a description file that holds administrative information for the InfoPak. This information comprises identifiers; versioning information; requirements for the InfoPak such as platform, hardware, and language; scheduling data, and other administrative information that describes how the InfoPak is to be processed.

BackWeb supports four channel types: BackWeb channels, web channels, file distribution channels, and CDF [39] channels. Each broadcaster (*BackWeb Polite Server*) can host one BackWeb channel, that can offer all of the functionalities available in BackWeb, and an unlimited number of the other channel types.
BackWeb supports pull distribution—via HTTP and the proprietary *BackWeb Transfer Protocol*—and push distribution, based on StarBurst's *Multicast File Transfer Protocol* [113] (see Section 3.1.4.5). The BackWeb Transfer Protocol (also denoted as BackWeb Polite Protocol) is based on UDP and is optimized for background communication. It tries to use network bandwidth economically and when not needed by other applications. It supports a checkpoint/restart functionality so that interrupted transfers can be resumed at the point of interruption.
Pull-based channels are queried every 5 minutes for updates by default (configurable). Pushlets can be executable files, Java applets, Java applications, and Netscape plugins. Differential updates are supported at a byte granularity and are transparent towards network disconnects. Backchannels are available by the concept of *up-stream data*, which supports the building of two-way push applications to interact with users. BackWeb clients can collect data from the user and send it to the server, which processes the data and can react accordingly. Users can filter channel data by type and content based on a wide range of available options.
Scalability ranges from medium to high depending on the distribution protocols and transport infrastructure used. BackWeb's transport system uses chained caches (called *proxy servers*). Receiver software has to be maintained manually, but upgrades are distributed by BackWeb Inc. as special InfoPaks to dedicated BackWeb servers. Certificates, digital signatures, and encryption are supported to ensure authenticated information and secure transmission.

3.3.3 Webcasting

Webcasting [112] is Microsoft's push technology. The receiver for channels is Microsoft's Internet Explorer (IE). Three types of Webcasting exist: *basic*, *managed*, and *"true"*. Basic webcasting means that IE performs a scheduled "sitecrawl" of a web site and checks whether new content is available. After having "subscribed" to a web site in this way, IE periodically checks the web site. If updates are encountered, IE notifies the user of the changes, or downloads the modified pages, and notifies the user (this depends on the user's configuration).
Managed webcasting is based on the Channel Definition Format [39] (CDF). CDF is based on XML [14] and allows the author of a channel to define the contents of a channel, its hierarchical structure, and an update schedule. The content of the channel is defined by a list of URLs. The structural description provided in a CDF file supports the hierarchical structuring of these URLs independent of their real structure on web servers. Thus CDF allows the author of a channel to define what parts of a web server are "webcast" and how they are logically grouped. The update schedule defines when the client should check for new content. The user can configure IE whether as to he/she wants to be notified about updates or the updates should be downloaded prior to a notification. Subscribing to a CDF channel means that IE downloads a CDF file from a web server and then acts according to the definitions given in the CDF file.
True webcasting means that CDF-based channels are actively "pushed" to subscribers. This relies on the integration of third-party software, for example StarBurst's *Multicast File Trans-*

fer Protocol [113] (see Section 3.1.4.5). Without third-party products the main communication paradigm of webcasting is to pull at user-configurable intervals (called *smart scheduled pull*). Since IE is used as the receiver, all supported web formats can be used. Thus pushlets can be any executable content that IE can deal with (Java, JavaScript, ActiveX, Windows applications, etc.). The update strategy is differential at the granularity of files. The user can choose between monitoring and downloading content changes and is notified of changes via IE (gleam on updated favorites) or via email (email HTML content). No explicit backchannels exist, but they can be implemented "outside" by means of Java, ActiveX, DynamicHTML, etc. The same applies to filtering of channel content. Users can choose the (parts of) channels they want to receive. Further filtering and personalizing can be made available by the channel provider.
Webcasting does not have a dedicated broadcaster or a transport system since it relies completely on the web infrastructure (web servers, caches, etc.). Thus it is scalable to the degree of the web itself. The receiver can be automatically updated via a special *software update channel*. Software update channels rely on OSD [170], which is a vocabulary to describe software components and their dependencies for deployment and has been submitted to the World-wide Web Consortium (W3C) to become a standard. OSD (see also Section 3.3.1) is a common standard of Microsoft and Marimba. Software update channels facilitate the automatic downloading and installation of software. Software packages sent via a software update channel can be authenticated. Other data is neither authenticated nor secured.

3.3.4 PointCast

PointCast [135, 136] is both a push system and an information provider. Only content coming from registered information providers can be broadcast via the so-called *Business Network*. The *Central Broadcast Facility* (CBF) is the central repository for PointCast network information. Three classes of channels exist: the *business network* that comprises the channels broadcast by CBFs, a freely configurable *intranet channel*, and a *connections superchannel*.
The intranet channel resembles the channels of the business network but can be freely configured and used by a company or organization as a local information channel. The intranet channel consists of configuration files, CDF [39] files that describe the content (groups) and its structure, and the actual channel content. The content distributed via the intranet channel consists of HTML files and ScreenPlay animations. Additionally, functionality similar to that of the business network channels, such as a screensaver or a ticker, can also be included.
The connections superchannel consists of connections, which are independent channels that can be created by anyone with a publicly accessible web site. A connection is defined by a CDF [39] file and is similar to a managed webcasting channel of Microsoft webcasting (see Section 3.3.3). Thus, basically any managed webcasting channel could be turned into a PointCast connection.
Channel data consists of web data formats and animations written in the *ScreenPlay* language, which can be considered a rather limited version of pushlets. Limited filtering of channel content is possible in that users can select predefined content classes and types within channels.
PointCast is highly commercial. Commercials can be attached to channel content and displayed by the receiver. They consist of animations that continuously run in the upper right corner of the

receiver and can only be filtered to a certain degree. It also includes a screensaver (*smartscreen*) and a ticker that can both be used for displaying information and advertisements.
Pull is PointCast's main distribution paradigm. The default pulling interval of clients is one hour but can be configured by the user. Push distribution (*alerts*) is available for intranets only (through multicast). PointCast has no dedicated broadcaster. The administrative and channel data are retrieved from web servers. Maintenance of this data is supported by a set of tools. Additionally, several publishing tools exist. No explicit information was found on the update strategy. PointCast has no backchannel concept.
A cache (called *cache manager*) is the only transport system infrastructure. An organization can have a primary caching manager (CM) and a set of second-level CMs. If multicasting is available the CM can act as a broadcaster and actively distribute notifications. The scalability of PointCast is mainly limited by the number of available CBFs, of which currently only a few exist. CBFs have limited support for load balancing: Requests to the data center can be forwarded to a *PointRouter* that redirects them to an *PointServers*. Receiver software can be updated automatically. Distributed data is neither authenticated nor secured.

3.3.5 WebCanal

WebCanal [95, 96] is a platform for *global information broadcast* on the Internet. It uses multicast push distribution based on the Light-weight Reliable Multicast Protocol [94] (LRMP) and the MBone [87]. LRMP is based on RTP [150] and a modified version of SRM [45] and was already described briefly in Section 3.1.4.4.
WebCanal consists of a *WebCaster*, a *WebTuner*, and several other tools. WebCanal is not only a push system but can also be used as a conferencing and presentation platform. It interacts with a web browser for displaying content, thus supporting pushlets on the basis of the executable content supported by the web browser used (Java, JavaScript, etc.). No explicit backchannels exist, but they can be implemented "outside" by means of Java, JavaScript, etc., depending on the web browser's capabilities. The same applies to filtering of channel content. However, a rather coarse topic-based classification of channels is supported. Updates are differential at a file granularity. The lack of explicit backchannels is partially compensated by the fact that WebCanal can also be used for symmetric two-way communication: Every receiver can act as a broadcaster. Since WebCanal relies on the MBone as its transport infrastructure, it has no special transport system. Its scalability therefore depends on MBone. This means that its scalability is currently low to medium but will grow as the scalability of MBone grows. Receiver software cannot be updated automatically, although WebCanal supports software distribution channels. Distributed data is neither authenticated nor secured.
WebCanal also is trying to establish open standards for push systems [96]. LRMP [94] provides a reliable multicast transport protocol on top of underlying connectionless networking protocols. WebCanal uses the Session Announcement Protocol (SAP) and the Session Description Protocol (SDP) for announcement of channels. WebCanal's version of SDP allows the description of channel properties and SAP announces channels via periodic multicast of channel properties. Its Multipoint Information Distribution Protocol (MIDP) provides the notion of *information channels*, which are a high-level abstraction of multicast sessions. Applications can build on MIDP

to provide guaranteed delivery to multiple users. Additional functionality to cope with network disconnects and “late-joined” users is also included.

3.3.6 Intermind

Intermind [180] is a pull-based push system without a dedicated broadcaster. Channels (administrative and content data) are available via web servers. Receivers (*Intermind communicator*) regularly check whether new content is available for a channel. A web browser is used for displaying channel content. Thus pushlets are supported on the basis of the executable content supported by the web browser used (Java, JavaScript, etc.).
[180] defines a framework for data and metadata exchange, i.e., exchange of *channel objects*, based on XML [14] and the Resource Description Framework [89] (RDF), between *communication nodes* as depicted in Figure 3.3.

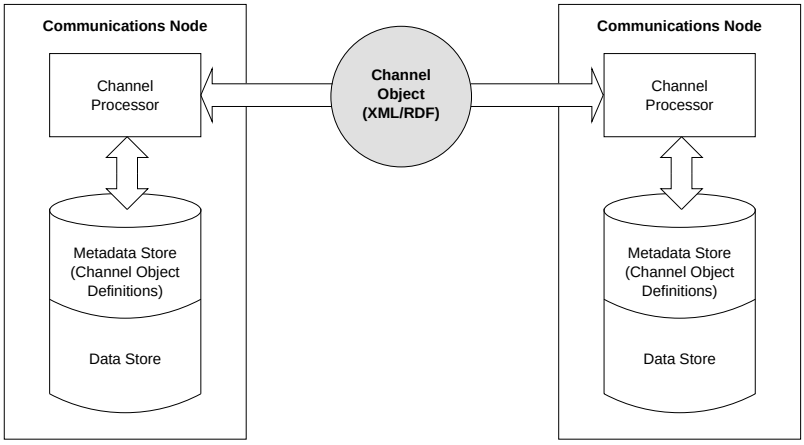

Figure 3.3: Intermind channel architecture [180]

Intermind owns a patent on push-like communication [78] that covers the following interaction scheme: Two nodes (the sender and the receiver) have persistent storage, communicate over a network, and exchange a control structure—the so-called *channel object*—which describes how to transfer updated information from the sender to the receiver, how to transfer feedback information from the receiver to the sender, and how to process the exchanged information.
A channel object encapsulates *data*, *metadata*, *methods*, and *rules*. Data refers to content that is to be sent over a channel. Metadata encapsulates profile elements for controlling channel processing. *Profile elements* are metadata attributes that describe what information to update, the data that is exchanged via a channel, how the data is transferred, and how to process the received data. A *profile* is the collection of channel metadata which describes a subscriber, a publisher, a subscriber, a channel, or a channel update. Methods define scripts or embedded objects for controlling channel processing and rules specify event-triggered rules for controlling channel processing. Methods and rules are optional.

Basically, channel objects are a higher-level concept to describe the information exchange between two nodes. Every node involved in a communication has a *channel processor* component that can process channel objects and act accordingly to receive and process data. Channel processors can store data and metadata and retrieve it from the persistent store of every communication node.
The channel object concept describes a flexible infrastructure for information exchange. However, it lacks exact definitions and specifications, and it is unclear to what extent this concept has been implemented in the Intermind software.
Channels (also denominated as persistent relationships in Intermind) can be defined with the *channel publishing tool*. The resulting descriptions and the data are placed on a web server where receivers can access it. No explicit information on the update strategy has been found. It is also unclear how backchannels are supported. Backchannels could be implemented using features offered by the web browser (Java, JavaScript, etc.).
Limited filtering is available at two levels: First, inside a channel the user can select topics to receive from predefined per-channel topics, and second, channels can be categorized according to user-defined categories. Intermind does not have a dedicated broadcaster or a transport system since it fully relies on the web infrastructure (web servers, caches, etc.). Thus it is scalable to the same degree as the web itself. Receiver software cannot be updated automatically. Distributed data is neither authenticated nor secured.
Intermind also offers a *Global Directory* for available channels. However, this directory only provides an alphabetically sorted list of channels without further functionality.

3.3.7 Minstrel

The component model of Chapter 2 can be used not only for comparison of push systems, but also as a basis for developing reference implementations for such systems. The Minstrel push system project described in Chapters 4 through 6 is such a reference implementation where the component model is used as an architecture for developing plug-compatible components for push systems and to devise an open protocol suite for Internet-scale content distribution. Minstrel is designed as a proof-of-concept implementation of the architectural model and serves as an extensible software platform for further investigations in the push area.
Minstrel uses the hybrid broadcasting paradigm that supports timely notification while requiring no special multicast infrastructure. The broadcaster pushes a “sample” (description of the available data, a small-size sample of the real data, and administrative data) to the subscribers of a channel; based on this information the subscribers may request the actual data as a “shipment” from the broadcaster. Besides the standard web data types, it supports arbitrary content types by its concept of agents that can be attached to the data and know how to process it. Minstrel supports pushlets, which are executed in a highly configurable Java Secure Execution Framework (JSEF) that supports definition of subtractive security policies and interactive security negotiation. Shipments define the granularity of updates and can either hold large archives or just the piece of data to be updated. The decision what granularity level to use is up to the user. Minstrel’s update strategy is equivalent to differential updates at a file granularity. Limited backchannel functionality is available by the use of Netscape Communicator as part of the re-

ceiver: It can be configured to use simple proxies that can communicate information back to the broadcaster.
Filtering can be done by exploiting the information that comes with every sample and shipment (description, keywords, administrative data). One of the main goals of Minstrel is high scalability, which is achieved by its transport system infrastructure. The transport infrastructure consists of repeaters, caches, and proxies. Minstrel's receiver supports flexible data access via its flexible data store unit. For displaying channel content it uses an off-the-shelf version of Netscape Communicator that is controlled via Minstrel's Netscape Remote Control Facility (NRCF).[2]
In Minstrel all transmitted information is authenticated before it is delivered. Encryption is not included as a dedicated package. However, SSL [128] can be used to provide encrypted and secure transmission. Additionally, Minstrel includes a flexible payment model that can be used for a variety of payment methods and business models, such as pay-per-view, volume-based, or flat fee.
Minstrel is implemented in Java and the protocols are based on RMI [66]. An open question is whether these paradigms will be competitive with socket-based communication in terms of efficiency and scalability.

3.4 Other Push Systems

A vast diversity of "push systems" exists. Many of them, however, do not exhibit the properties defined and described in Chapter 2 as the systems discussed in Section 3.3. This section gives a concise overview of other push-like systems and approaches. They may differ considerably in terms of functionality, completeness, and application domain. Although they are not described in detail, they are mentioned here to provide a more complete picture of the "push market."
The first rudimentary push approach was devised in 1992 in Netscape Navigator 1.1 with its **dynamic document** concept. Its basic ideas are *server push* and *client pull*. With server push the server sends data which is displayed by the browser, but the connection between server and client remains open. Later the server may continue to send other pieces of data to the client. Client pull automates reloads: The server sends data including a `Refresh` directive specifying a time delay and a URL in the HTTP response or in the document header. After the given delay the client loads the document specified by the URL.
Salamander [104] is a push substrate for real-time data distribution over the Internet. It intends to provide a push infrastructure for applications. The distribution infrastructure consists of collaborating servers, organized in an arbitrary topology, that receive and forward channel data (servers are so-called distribution points). Salamander's communication paradigm is push. Clients, which can both act as publishers and consumers, connect to a specific server in the infrastructure to publish or receive data. Salamander distinguishes two types of subscriptions: anonymous push and negotiated push.

[2]The implementation of NRCF actually can be adapted easily to work with other web browsers, too. Basically, any web browser that supports a certain extent of "remote control" functionality via Java applets, such as Microsoft's Internet Explorer, which was also evaluated, could be used as Minstrel's display unit. Netscape Communicator was given the preference, however, and used in Minstrel, since it also is available for non-Microsoft platforms.

With anonymous push, clients register persistent queries (lists of text attribute expressions) with a server. These queries determine the kind of data the client wants to receive and provide a powerful filtering mechanism. Since every client can receive a data flow tailored to its interests, channels are called virtual channels in Salamander. Data that is distributed via Salamander is described by text attribute lists. The data itself is opaque to the system (this is closely related to Minstrel's approach). The unique property of client queries is that they are matched against both future objects and objects that already are in the lightweight temporal database that is part of the system. The update strategy of this database can be viewed as incremental at the granularity of objects. The database provides lightweight persistency.
Negotiated push provides a backchannel functionality: Subscribers can contact publishers and provide feedback, for example, by asking them to begin data distribution or to modify a supplier's data flow at the source. Additionally, a notification service exists to propagate various system events among clients (group membership, etc.).
Salamander also supports application-level quality of service policies. These policies are a way to tailor the available resources to best fit the user and application by exploiting application domain knowledge. Typical policies define data degradation, data conversion, or synchronization. Salamander provides good scalability and throughput and has been deployed in two big research projects that needed push functionality. However, Salamander does not support pushlets and does not address security issues. Only a limited authorization scheme for accessing the infrastructure exists. Receiver software has to be updated manually.
Common Datacast Architecture [72] (CDA) is not a push system per-se but an approach to developing a general architecture for customized information subscription, publication and dissemination. It aims to integrate existing push systems into a uniform distribution platform that can be used by information providers. The so-called channel builder allows providers to define subject-based information channels which it the transmits via various channel transmitters (push systems). CDA exists as a proof-of-concept prototype.
Java Message Service [65] (JMS) provides an API for accessing enterprise messaging systems from Java programs. JMS supports point-to-point and publish-subscribe communication models and can work over different transport providers such as TIB/Rendezvous (see Section 3.5.1.1). Several JMS implementations from different vendors are available.
Web Transporter [156] is a toolset for software deployment and maintenance. It offers user-initiated pull and scheduled push. Web Transporter Agent allows administrators to deliver files on a predefined schedule. Web Transporter Server provides software maintenance functionality, authentication, and management functions. Users are authenticated prior to software delivery. Access limitations based on user name and group membership are possible. Secure communication is supported based on SSL [128].
BusinessWare [172] provides a communication platform for transparent application integration at the business level. It offers event-driven, publish-subscribe messaging via channels that communicate related data to a common set of subscribers. BusinessWare includes infrastructure components to support scalability and fault tolerance (cache channels, replica channels). Secure and authenticated communication is supported. Sophisticated analysis tools are available that provide real-time decision support based on the received information.
The **Basic Lightweight Information Protocol** [81] (BLIP) is a proposed protocol that aims at

providing a language- and platform-independent wire protocol for high-speed, large-scale, reliable, topic-based messaging. It is designed around message queues: Clients subscribe to servers that hold message queues on their behalf. Whenever the client connects to the server it receives its queued messages. While connected to the server, the client gets its messages immediately. This proposal is in a very early stage and closely resembles the mailbox functionality known from electronic mail servers. It is unclear whether it will be pursued further.
Section 3.3 provided a structural classification of representative push systems. An application-oriented classification is suggested in [15]. Push systems are classified as application distributors, content aggregators, platform providers, or business-dedicated content aggregators. An application distributer provides a distribution platform for software deployment. A typical example is Castanet (see Section 3.3.1). Content aggregators are news and information sources. They gather content, convert it to a specific presentation format, and deliver it to subscribers. A typical example is PointCast (see Section 3.3.4). Platform providers provide platforms for content providers to deliver their content to users. They differ from content aggregators in that control of the distribution is in the hands of the content provider. A typical example is BackWeb (see Section 3.3.2). Business-dedicated content aggregators do not provide services to individual users, but concentrate on business and enterprise-wide implementations of push.

3.5 Related Paradigms

The diffusion of the Internet has given rise to a number of novel distributed programming paradigms. Among these, push systems, mobile code, and event-based systems are closely related. This section discusses the relationships between these three systems, their commonalities and distinguishing properties.

3.5.1 Event-based Systems

Event-based systems define a new style for the construction of (distributed) applications based on the notion of events. In such systems, components interact by generating and receiving events. Components declare interest in receiving specific events and are notified on occurrence of those events. This pattern supports a highly flexible interaction between loosely-coupled components [26].
The architectural model is well-developed for local area networks. In a large-scale, heterogeneous setting like the Internet, however, new and adapted technologies are needed since many of the premises of a LAN do not hold at the Internet-scale. A design framework for Internet-scale event-based systems is presented in [144] that suggests a seven-dimensional design space: object model, event model, naming model, observation model, time model, observation model, and resource model. Some classifications of event-based approaches are given in [26]. Design issues of event-based architectures are discussed in [16]. The special problems of Internet-scale event observation and notification are overviewed in [145]. [179] presents recent developments and approaches in the area of event-based systems.

Push systems and event-based systems are closely related. In fact, it is not always clear where to draw the dividing line. Distinctive properties exist, however. Table 3.3 lists the main differences between the two paradigms.

	Push Systems	Event-based Systems
Purpose	timely data distribution	event notification
Participant roles	asymmetric	symmetric
Advertisement policy	simple advertisement (channel)	expressive advertisement language
Subscription policy	simple subscription (channel)	expressive subscription language
Frequency of events	low to medium	high
Number of events	low to medium	high
Payload size	large	small
Producer/consumer interconnection	static channels and static producers	dynamic binding to producers
Event grouping	channel	event patterns
Filtering	reduce data transmission requirements	reduce number of events

Table 3.3: Push systems vs. event-based systems

Whereas the purpose of push systems is the timely distribution of data to consumers, event-based systems focus on notification of events. The roles of participants differ considerably: Push systems have two distinct groups—event producers (broadcasters) and event consumers (receivers)—while in event-based systems anyone can produce and consume events. The announcement and subscription of new information is simplified in push systems since they can rely on the channel concept that provides a tighter coupling between producers and consumers while still providing some flexibility. Event-based systems, on the other hand, have only a very loose coupling between producers and consumers and therefore must have powerful mechanisms for event selection (both for the provider and for the consumer).
The number and frequency of events in push systems will be limited by content transmission rates and thus be at a moderate level. Event-based systems in contrast are targeted at possibly very high event rates. Closely connected with this are the payload sizes: While the size of the payloads transmitted in push systems can be quite large (since they are information-oriented), an important design criterion in many event-based systems is to minimize the size of events. Due to the channel concept, the interconnection of producers and consumers in push systems is rather static. Consumers are likely to receive a fixed set of channels from a set of producers with little change. Though consumers are notified of new channels, for example via a meta-channel, subscription and unsubscription will occur infrequently once a satisfying profile of interests exists.

Channels also provide an implicit mechanism for event grouping. A channel will offer "events" of a certain quality only (e.g., weather forecast channel). Loose coupling in event-based systems will lead to more dynamic interconnections. Event producers can be mobile and change frequently. For example, in a distributed system one can imagine a network administration mobile agent that travels among networking components and generates administrative events. Event-based systems are intended to have sophisticated event grouping mechanisms called event patterns. Consumers are able to group events by patterns, e.g., *XY*Z*, meaning event *X* followed by zero or any number of events *Y* followed by event *Z*, and receive a single notification upon occurrence of a pattern. Though the usefulness of this mechanism is undoubtedly high it adds considerable complexity to the implementation of such services. Ordering (timely, causal, etc.) of distributed events is a highly complex research area that still needs further investigation. Patterns operate on the level of events, while filters operate on content-specific information to select events for notification. In push systems, filters help to reduce data transmissions, while in event-based systems the goal is to cut down on the number of events.
While other authors have argued in favor of regarding push technology as a special case of a more general *Internet Notification Service* (INS) [13], this comparison shows that, despite their similarities, event-based and push systems are distinctly different models of distributed information systems because of their differences in foci, requirements, and applied concepts.
The following sections briefly showcase some event-based systems to demonstrate the variety of approaches in this area.

3.5.1.1 TIB/Rendezvous

TIB/Rendezvous [164] is a commercial infrastructure for creating and maintaining large, distributed event-based applications [26]. It is based on TIBCO's *information bus* (IB) technology and supports a publish/subscribe communication pattern between producers and consumers. Publish/subscribe interactions are event-driven, in producer-consumer direction only and mostly one-to-many. Consumers place a standing request for data by subscribing.
TIB/Rendezvous utilizes so-called subject-based addressing and subject-based multicasting. Information is published (sent) on a subject name and interested consumers can subscribe to these subjects and receive the according messages. This approach differs from other approaches that usually use addresses such as IP addresses to send information to, which do not carry information on the data transmitted. TIB/Rendezvous thus facilitates a content-based addressing scheme which is very attractive for push systems and event-based systems. Additionally, data is multicast to conserve network bandwidth and support efficient delivery to high numbers of consumers. TIB/Rendezvous' multicast communication is reliable.
Applications receive messages by *listening*, which associates a callback functions with a certain subject name. Upon receipt of an appropriate message TIB/Rendezvous sends it to the subscribed party by dispatching it to the appropriate callback function. The concept of subject-based addressing also supports location transparency, since messages do not hold physical addresses but subjects. Thus producers and consumers can be freely migrated or reconfigured.
Besides the publish/subscribe communication model, which is the most interesting one for push systems, two other models are supported: standard request/reply interactions that involve one

producer and one consumer, and broadcast request/reply interactions where multiple producers receive a request and send responses.
The event-dispatching in TIB/Rendezvous is done in a three-level hierarchy. Every participating site runs the TIB/Rendezvous daemon that is in charge of sending, receiving, and filtering messages for consumers running on that site. This daemon also delivers messages to any TIB/Rendezvous application on the same network via broadcast messages.
The TIB/Rendezvous routing daemons are responsible for delivery between networks. Two specialized routing daemons exist: The wide-area routing daemons link distant sites, whereas the subnet routing daemons connect collocated networks, e.g., multiple subnets of a single organization.
TIB/Rendezvous offers a powerful substrate for push systems to build on. Under a broad definition of push systems, it could even be called a push system itself. It supports subject-based addressing which allows the consumer to state its interest in a meaningful way. Its distribution protocols are efficient and reliable. However, it is unclear whether its infrastructure would scale to Internet size and it lacks several properties of push systems as defined in Chapter 2. Additionally, its protocols and specifications are proprietary and not open to third-party software developers. Nevertheless it is a leading commercial platform and has gained widespread deployment.

3.5.1.2 Keryx

Keryx [13] is a notification system which tries to set up a general, standardized *Internet Notification Service* based on a language- and platform-independent event distribution infrastructure. Keryx's distribution infrastructure consists of so-called Event Distributors (EDs) to decouple event producers and consumers. EDs can be organized in an arbitrary topology. Parties interested in notifications subscribe with an ED and provide a filter that only succeeds for events that the party is interested in. An ED matches every event it receives against the registered filters and conditionally forwards notifications to the interested parties. It also forwards events to other EDs. This configuration provides an infrastructure that supports routing by event content.
Notifications do not include content themselves. For content distribution a hybrid push-pull scheme is suggested: Clients describe events they want to be notified of; when such an event occurs, clients are actively notified (push); a notification can hold information where the client can retrieve the actual content (pull). This is related to the hybrid approach the Minstrel system takes (see Chapter 4).
Notifications are delivered via the so-called Event Transfer Protocol (ETP). ETP is based on TCP. Alternatively notifications can be delivered via SMTP [137]. At present, multicasting is not supported; notifications are delivered individually to each subscriber.
Keryx focuses especially on the description and filtering of events. A notification is structured information that describes an event. It contains a description of the type of the event together with qualifying information. Notifications are represented in a self-describing, human-readable, textual data syntax called Transfer Syntax (TS), which is also submitted as an Internet Draft under the name Self-Describing Data Representation [101] (SDR). It is designed as a transfer syntax for loosely-coupled distributed applications that do not share a schema for the exchanged

data. TS's data model supports the representation of structured information using atomic values, lists, and maps as its basic data types. Every TS value is associated with a tag. Figure 3.4 shows an example event definition in TS.

```
event: {
  type (emit),
  content {
    type (stock-price price-change),
    stock-symbol  "HWP",
    stock-price   "$65",
    exchange-name "NYSE"
  },
  system {
    message-id 4711
  }
}
```

Figure 3.4: A Keryx event specification

Keryx supports the definition of event filters through its Default Filtering Language (DFL), which provides a language-independent mechanism to express simple predicates over TS values. Clients can include such predicates in their subscriptions to specify the kind of events they are interested in and want to receive notifications of. DFL supports equality and relational tests, logical operators, and quantifying filters over lists. It consists entirely of expressions. For example, the predicate `(test (content stock-symbol) (equal "HWP"))` defines a filter that matches the event specified in Figure 3.4.
Keryx is an interesting approach that provides powerful content-based event delivery and expressive event filtering mechanisms. Its distribution infrastructure, however, seems to need improvement before it can scale to large numbers of subscribers over the Internet. Keryx is used as the communication infrastructure in a Distributed Virtual Environment (DVE) system [181].

3.5.1.3 Java Event-based Distributed Infrastructure

The Java Event-based Distributed Infrastructure [25, 26] (JEDI) is an object-oriented infrastructure for event-based systems. JEDI is based on the notion of *active object* (AO), which denotes an autonomous computational unit performing an application-specific task and interacting with other AOs by means of events. A JEDI event is an ordered set of strings: an event name followed by event parameters. AOs subscribe to an *event dispatcher* (ED), which is in charge of disseminating events to AOs. In the subscription the AOs declare the classes of events they are interested in. AOs can also subscribe to *event patterns*, which are a simple form of regular expressions over events. JEDI guarantees a partial order among events: it ensures that events generated by a given source are delivered to all recipients in the order in which they have been generated.
A *reactive object* is a special form of an AO: An object registers with the ED and waits for events; upon receipt of an event it performs some computation and waits again. JEDI also supports mobility of such objects: A reactive object can decide autonomously to move to a different host.

This migration is transparent towards event delivery; even though the object has moved, it will continue to receive events, and no events are lost during the migration.
To support scalability the event dispatcher is available in a distributed version consisting of cooperating *dispatching servers* (DS). DSs are organized in a tree topology and use a hierarchic strategy for subscription and event distribution: Subscriptions are propagated up in the tree, so that all ancestors of the DS, which has accepted the subscription, receive it; conversely, when a DS receives an event it only needs to dispatch the event to its parent, its decendant DSs that have subscribed to this event (pattern), and its directly connected AOs. This strategy efficiently operates for a variety of scenarios and tries to balance the number of required messages.
Distributed software components frequently need to interact by generating and consuming events. The event-based architectural style has been proven viable for a wide range of applications due to its highly flexible interaction pattern between loosely-coupled components. JEDI provides a substrate that facilitates the implementation of such systems. It has been successfully used as underlying infrastructure in the implementation of the OPSS workflow management system [26] which is part of the ORCHESTRA system [32]. [26] also presents a classification framework for event-based systems and compares representative event-based infrastructures based on this framework.

3.5.1.4 CORBA Event Service

Also industry standard architectures like the Common Object Request Broker Architecture [130, 155] (CORBA) are starting to incorporate support for the event-based paradigm. CORBA is a standard software architecture from the Object Management Group (OMG) for the component-based development and deployment of applications in distributed, heterogeneous environments using the object-oriented paradigm.
The CORBA Event Service [131] (ES), which is based on the event service described in [1], defines a set of interfaces and an underlying infrastructure that allow objects to communicate events to each other via so-called *event channels* which provide the distribution infrastructure for events. *Suppliers* connect to an event channel to send events to it and *consumers* connect to an event channel to receive events from it. The concept of an event channel decouples communication between suppliers and consumers. To suppliers an event channel acts as a consumer; to consumers it acts as a supplier. All events sent from a supplier to an event channel are transmitted to every consumer of that event channel. Multiple event channels can be used in parallel so that objects can group events. Consumers only connect to the channels they are interested in (this is similar to the channel concept of push systems).
The ES supports two communication models for suppliers and consumers: *push* and *pull*. In the push model the suppliers control the flow of data by pushing it to consumers; in the pull model the consumers are in control of the data flow by pulling data from the suppliers. A push supplier actively sends events to the event channel, while a pull supplier is requested for new events by the event channel at regular intervals. A push consumer passively waits for events from the event channel, while a pull consumer must check with the event channel at regular intervals whether new events have arrived. Since suppliers and consumers are decoupled these communication models can be mixed; for example, a push supplier can have both push consumers and pull

consumers. How events are distributed is not specified in the CORBA Event Service standard. So far, broadcasting/multicasting of events is not standardized.
This architecture provides a basic infrastructure for event-based applications. However, it is not complete and lacks many of the aspects of event observation and notification defined in [144]. [145] argues that the CORBA Event Service has several shortcomings: Its object model is constrained by CORBA's model, an event is only a message passed from one object to another as a parameter of an interface method, and the content of an event message is not defined, so receiving objects must "know" the event message structure to process it in a meaningful way. Standard semantics and protocols for event channels are lacking. With this view of an event, a naming mechanism and an observation mechanism are unnecessary (though these mechanisms are important properties if addressing the full problem space of the event-based paradigm) and the identification of event patterns is up to the consumers and not available from the infrastructure. It is not possible for a consumer to specifiy interesting events at a fine granularity by means of detailed specification of parameters and wildcards [103]. Only very limited event filtering is supported. In summary, several important abstractions and services, such as locating, advertising, and filtering of events or matching event patterns, are not available from this model. Applications based on this model have to provide their own solutions for these issues.
Nevertheless, several higher-level services, such as the TINA Notification Service [163], have been implemented on top of the CORBA Event Service. The COrba-Based Event Architecture [103] (COBEA) attempts to remedy many of the problems described above. COBEA is based on the Cambridge Event Paradigm [8] and extends the CORBA Event Service with the *publish-register-notify* mode; fine-grain, parameterized event filtering; fault-tolerance; access control; support for composite events; direct/indirect notification of events and dynamic addition of user-defined event types.

3.5.1.5 Notification Service Transfer Protocol

Though not providing a true event-based infrastructure, the Notification Service Transfer Protocol [30] (NSTP) is included here to demonstrate that event-based concepts are inherently present in many applications and systems that do not define themselves as being event-based. This speaks for the widespread applicability of the event-based architectural style.
NSTP is an infrastructure for building synchronous groupware. In such systems two or more parties collaborate at geographically dispersed locations. Participants must share a consistent state to promote the illusion of working together "at the same time." In terms of an event-based system this state consistency problem means that changes to the shared state must be delivered to all participants in a timely manner, i.e., participants must receive notifications of events and update their view of the shared state accordingly.
In NSTP the state of a collaboration is stored at a *notification server* and can be changed by the participants. Changes are communicated to all participants. The shared state is modeled in terms of *Things* and *Places*. Things consist of a mutable value and immutable attributes. An event in this model can be viewed as the modification of a Thing's value. A Place is a container for Things. Places are described by Facades. A Facade allows a client to decide whether a Place is of interest and provides the client with means to interpret the Place, for example, Java code

that allows the client to interact with the Place in a meaningful way (comparable to pushlets or agents).
Each Thing is in exactly one Place and cannot move between Places. Clients can associate themselves with one or more Places, i.e. become participants. If a client changes a Thing or creates a new Thing, all clients in this Place receive a notification and can react accordingly. This interaction pattern implements a simple event observation facility (a very limited version of JEDI's reactive objects as described in Section 3.5.1.3).
The advantage of NSTP is that it models notification in terms of a protocol. This means that it focuses on the coordination machinery rather than particular applications. Every entity that understands the protocol can interact with the system. The notification server is rather flexible and simple to implement, since it does not have knowledge about the semantics of the Things it handles. It simply treats them as chunks of uninterpreted bits and delivers notifications if they change. However, NSTP does not exhibit any further properties that are required for a fully-fledged event-based system.

3.5.2 Mobile Code

Program code used to be bound to a certain processor/architecture/computer. In contrast to this, the intention of the mobile code paradigm is to have code travel around networks and computers. So-called mobile agents [178] are an interesting approach for addressing information discovery, brokering, and scalability problems of information systems.
Channel content in a push system can consist of executable code that is to be executed at the receiver. Thus push systems must address similar issues as pure mobile code systems, although on a simpler scale, since some of the problems in mobile code systems do not arise for push systems (routing of agents, intelligent agents, protection against tampering of agent data, etc.). The main intersection of issues is in protecting host systems against malicious code and controlling access to host system resources.
A push system can be seen as a mobile code system and vice versa: A push system that distributes pushlets is a special case of a mobile code system. If a push system distributes pushlets and every receiver in the push system is also a broadcaster to route and forward pushlets, then this is similar to a mobile agent system. A mobile code system, on the other hand, can be used to actively transport information to users and thus can serve as a push system. One such approach is described in [51]. The essential difference between the two systems is in intent of use: Push systems are data-centric, focusing on efficient dissemination of information, whereas mobile code systems are functionality-centric, dealing with the distribution of computation to reach a defined goal.
Good overviews of existing mobile code technologies and paradigms are given in [102] and [171].

Chapter 4

The  MINSTREL Push System: Broadcast Communication

Minstrel is a push system that adheres to the component and communication model described in Chapter 2. It is intended to meet the following goals:

Scalability. Minstrel is designed to scale to large numbers of users while trying to keep network traffic at a reasonably low level. The Minstrel dissemination infrastructure and protocols are designed to support this major design issue.

Real push. The majority of the systems described in Chapter 3 do not use an active broadcasting strategy but require the receiver to poll at regular, configurable intervals (see 2.2.2.1 for techniques used to implement a broadcasting strategy). This strategy scales well to large numbers of users and can be employed without changes to the current standard Internet infrastructure, but solves only part of the problems described in previous sections. The remaining systems apply active broadcasting strategies but require specialized multicasting infrastructures, like MBone [40], TIB/Rendezvous [164], or StarBurst's Multicast File Transfer Protocol (MFTP) [113, 158], which currently lack wide-spread deployment. Minstrel uses a hybrid broadcasting strategy (see Section 4.2) that actively distributes information via MADP (Minstrel Active Distribution Protocol). MADP disseminates small-size records (*samples*) that inform the receiver about the availability of information and its attributes. The attributes define the content type, size, price, etc. of a piece of information. On the basis of these attributes, the receiver (user) can decide whether to request the information (via MRRP) or ignore it.

Authenticity and integrity of information. For deployment of push systems in commercial environments this is of premier importance, especially if the information received is to form the basis of business decisions. Receivers must be guaranteed that the information they receive comes from an authenticated source, and has not been tampered with, while broadcasters may want to ensure that only authenticated receivers access their channels. Minstrel supports information authenticity and integrity by its Minstrel Data Lock (MDL) subsys-

tem (see Section 6.1.1). Every piece of information that is transferred via a Minstrel channel is digitally signed.

E-commerce. While currently most information available via the Internet is free of charge, this will change dramatically as soon as the "electronic marketplace" becomes reality. Adding e-commerce facilities to existing systems is complicated and not always possible. Thus Minstrel is designed to support e-commerce from its initial stage and to provide support for various payment methods and business models. As a proof of concept, Minstrel supports a *pay-per-view* business model using the Millicent micro-payment protocol [54] (see Section 6.2) as the payment method.

Executable content. Minstrel supports the dissemination of executable content that is intended for execution at the receiver (*pushlets*). Pushlets are Java [4] applications that run inside a secure execution environment to protect the receiver from malicious code. This secure environment—Minstrel's Java Secure Execution Framework (JSEF)—is described in Section 6.1.3. Like all other Minstrel information, pushlets are authenticated.

Open protocols and interfaces. One of the key problems of many other push systems is that they cannot interoperate, since their specifications and standards frequently are not publicly available. Accessing n push systems—regardless whether as a content provider or an end user—typically requires n incompatible broadcaster/receiver/transport platforms that transport possibly identical data in n incompatible formats with incompatible protocols. Standardization, however, is important to support wide-scale usage and third-party products. Thus Minstrel's protocols and interfaces are open to the public.

Open and extensible architecture. Minstrel is designed to support quick adoption of new technology in a flexible way.

This chapter is organized as follows: Section 4.1 starts with an overview of Minstrel's architecture and is followed in Section 4.2 by a general description of Minstrel's broadcasting strategy. Section 4.3 then presents the details of channel subscription and Section 4.4 describes the key data structures used in MADP and MRRP. Section 4.5 overviews how received channel content is processed. Section 4.6 gives a detailed analysis and discussion of Minstrel's broadcasting strategy based on a concrete scenario. Finally, Section 4.7 presents the main protocols—MADP and MRRP—in detail.

Subsequent chapters present other main aspects of the Minstrel system and how it addresses those goals that are not discussed in this chapter. Chapter 5 gives detailed descriptions of Minstrel's main components (receiver, broadcaster, and BDC) and Chapter 6 continues with security issues (authentic information, confidentiality, and mobile code security) and describes Minstrel's support for business models and electronic payment.

4.1 Architecture and Overview

Minstrel's architecture follows the component and communication model for push systems described in Chapter 2. Figure 4.1 depicts Minstrel's architecture at a coarse level:

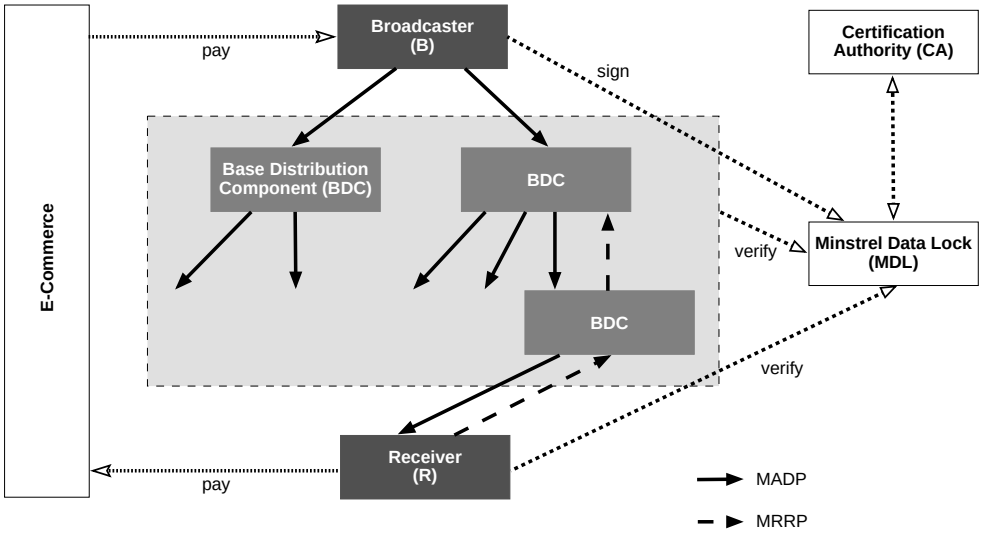

Figure 4.1: Minstrel's architecture

Minstrel combines the *primary broadcaster* and *simple broadcasting* approaches described in Section 2.2.2: it has a primary broadcaster and relies on functionalities of the transport system for the distribution of channel content to large numbers of users. Broadcasters provide a flexible interface for *information sources* to supply channel content that is to be distributed. Broadcasters can offer multiple channels that they disseminate via a hierarchical transport system consisting of *base distribution components* (BDCs). This is Minstrel's approach to the design goal of scalability. BDCs can operate as repeaters, caches, or proxies (see Section 2.2.4), depending on their configuration and the structure of a specific transport system instance.

Receivers are the software components at the client side that allow human users to access channels and interact with the Minstrel system. If we abstract from the transport system that is designed to be transparent towards broadcasters and receivers, broadcasters and receivers are the parties that interact in the Minstrel system. However, if we do consider the transport system, BDCs can also be modeled in terms of broadcasters and receivers: Towards their feeding entity—a broadcaster or another BDC at a higher stratum in the hierarchy—they operate in much the same way as a receiver (subscriber of a channel); downstream, towards other BDCs or receivers, on the other hand, BDCs act similarly to broadcasters.

Minstrel uses a hybrid broadcasting strategy as described in Section 2.2.2.1: the broadcaster pushes a *sample* (description of the available data, a small-size sample of the real data, and administrative information) to the subscribers of a channel; on the basis of this information, the subscribers may request the actual data as a *shipment* from the broadcaster. *Samples* are distributed using the Minstrel Active Distribution Protocol (MADP); receivers request channel

content via the Minstrel Receiver Request Protocol (MRRP). Both protocols are based on Java RMI [66].
To support payment and business models, Minstrel offers a flexible and generic framework which decouples business models from underlying payment methods. At the moment Minstrel supports a pay-per-view business model using the Millicent micro-payment protocol [54] as payment method.
The Minstrel Data Lock (MDL) is Minstrel's infrastructure to provide authentication and integrity of channel content. MDL assures that circulated information comes from an authentic source and has not been tampered with.
As noted above Minstrel follows the component and communication model described in Chapter 2. Figure 4.1, however, deviates slightly from the model depicted in Figure 2.2:

- Figure 4.1 shows only one broadcaster and one receiver. This does not truly reflect Minstrel's architecture. Minstrel supports an arbitrary number of broadcasters and receivers, broadcasters can transmit an arbitrary number of channels, and receivers can obtain an arbitrary number of channels from an arbitrary number of broadcasters. The simplified architecture as shown in Figure 4.1 was only introduced to streamline the presentation of Minstrel and its concepts (without constraining generality).

- Figure 2.2 uses the notion of channels while Figure 4.1 shows the conceptually lower level of broadcasting protocols. This was done intentionally since the interesting problems of channel transmission must be solved at this lower level.

- Minstrel's standard transport system topology is a tree.

- Figure 4.1 shows supporting infrastructures (E-Commerce, MDL, CA) that were only implicitly described in Chapter 2. These components, however, must be addressed explicitly in the description of Minstrel since they are necessary to meet some of the requirements listed in Chapter 2.

4.2 Minstrel Broadcasting

Minstrel uses a two-step hybrid broadcasting algorithm:
Step 1 (push part)
Whenever new channel content becomes available, the broadcaster sends out a description of it. This is called a *sample*. A sample holds a description of the data, possibly a small-size version of the real data,[1] and administrative data. The broadcaster sends the sample to its directly connected BDCs, which further propagate the sample to their directly connected BDCs until the sample reaches the receivers that have subscribed to the channel holding the sample. This step is handled by the Minstrel Active Distribution Protocol (MADP).

[1] For example, if the offered data is an image, this could be a low-quality, small-size "thumbnail" of the image.

Step 2 (pull part)
On the basis of the information in the received sample, the receiver decides whether it is interested in the actual content. This decision can be made automatically if the user has defined appropriate rules for the receiver software or may require user interaction. For example, the user may want to receive weather maps from a weather channel automatically if the maps are free of charge and smaller than 200KB; if the user would have to pay for a map s/he could configure his/her receiver not to request it; if the map is bigger than 200KB, the user can require the receiver to ask explicitly whether the map should be requested. If the decision to request the content is positive, the receiver requests it from its *service agent* (SA). An SA is a conceptual role and denotes an access point of a receiver to the Minstrel infrastructure. BDCs serve as SAs for receivers. All communication between the receiver and the Minstrel system is carried out via an SA (channel subscription, sample distribution, content request, etc.), which interacts with the Minstrel system on behalf of the receiver and vice versa. Upon receipt of the receiver's request, the SA tries to fulfill it. If the SA can find the requested content (the so-called *shipment*) locally, it immediately sends it to the receiver. Otherwise the request is propagated to the SA's service agent, which can be either another BDC or a broadcaster. This continues until an SA can satisfy the request, resulting in the distribution of the content to the receiver as the initiating requester. This step is handled by the Minstrel Receiver Request Protocol (MRRP).

4.2.1 An Example Broadcast

Figure 4.2 shows an example of the interactions during a broadcast:

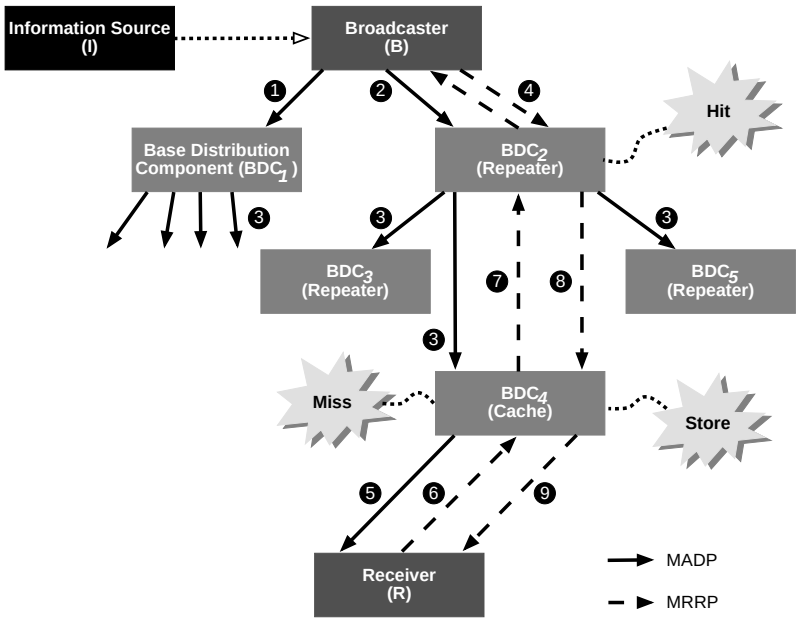

Figure 4.2: Minstrel hybrid broadcasting

Given that for a certain channel *C*, a new piece of information becomes available, the information source *I* sends the new content and its description to the broadcaster *B* and requests *B* to distribute it via channel *C*. On the basis of the information provided by the information source, *B* creates a sample *S*. To find out where to distribute *S* to, *B* consults its subscription database for a list of its subscribers to channel *C*.
It is important to note, that *B*'s subscription database only holds its local (directly connected) subscribers for channel *C*. Minstrel has no centralized subscription database. Instead, the subscription database is distributed among the disseminating nodes (broadcasters and BDCs). A node wishing to receive a certain channel subscribes to its SA (broadcaster or BDC) for this channel and the SA enters this subscription into its local database. The subscription is not forwarded to other nodes. Section 4.3 describes the notion of subscription in detail.
Thus in the case of Figure 4.2, *B* will find out that for channel *C* two subscribers exist: BDC_1 and BDC_2. Conversely, *B* is the SA of these two BDCs regarding channel *C*. As the first step in the broadcasting process *B* sends *S* to these BDCs (steps 1–2). This can be done iteratively by contacting each subscriber separately or via multicast. In the current implementation Minstrel uses the first strategy. Though it is an advantage, multicast is not a requirement for this step since samples should be rather small (a few kB) and only a limited number of direct subscribers may exist. If this number exceeds a configured threshold (e.g., 30), an additional layer should be introduced.
After having received *S*, BDC_1 and BDC_2 in turn distribute it to their subscribers in the same way described for *B* (step 3): Find the subscribers and send *S*. This step is repeated for every layer in the transport system. All steps described so far are done via the Minstrel Active Distribution Protocol (MADP).
Step 4 may occur concurrently to step 3: BDC_2 is a repeater and thus immediately requests and stores the actual content *S* was referring to. By doing so repeaters are preloaded with the channels' contents and offer the same data as the broadcaster but are "closer" (in terms of network properties) to the receiver. Thus a repeater has similar broadcasting capabilities as a broadcaster but cannot insert new content into a channel. The request for the content and its transmission to BDC_2 are done via the Minstrel Receiver Request Protocol (MRRP).
As in MADP, the two communication partners of MRRP are at adjacent layers of the transport hierarchy. These layers are defined by the subscription relation (see Section 4.3). For example, *B* would not send *S* to BDC_3 since BDC_3 is not subscribed at *B* und thus is not adjacent to *B*. BDC_3 on the other hand is a repeater like BDC_2, but would not request the content described by *S* from *B*. Instead it would request it from an adjacent component, in this case BDC_2, since it is subscribed to this component (in other words, BDC_2 is BDC_3's SA).
BDC_4 offers the same functionality as BDC_2 but is configured as a cache. Therefore it does not automatically request the content *S* describes, but of course actively distributes samples like a repeater. Concurrently to step 4, the further distribution of *S* continues until it actually arrives at a receiver (step 5). This is the end of the active distribution step of Minstrel's broadcasting algorithm.
Now the receiver *R* can decide whether it is interested in the content *S* describes. To allow a reasonable decision, *S* must carry sufficient information, which must be ensured by the information source. This decision can be made automatically on the basis of user-defined rules or may

require user interaction.
If *R* decides to obtain the content (a *shipment* in Minstrel's terminology), phase 2 of the broadcasting cycle starts. *R* requests the shipment from its Service Agent (SA) in step 6. In this example *R*'s SA—its access point to the transport system regarding channel *C*—is BDC_4. Generally speaking, SAs model the concept of an access point to a higher layer in the Minstrel infrastructure. For example, BDC_2 is BDC_4's SA and *B* is BDC_2's SA.
BDC_4 is configured as a cache and does not yet have the requested shipment (Miss). To fulfill *R*'s request it in turn propagates the request to its SA (step 7). BDC_2 is configured as a repeater and has thus already requested the shipment from its SA in a previous step (step 4). If the transmission of the shipment to BDC_2 is already finished (Hit), BDC_2 can immediately send it to BDC_4 (step 8). Otherwise BDC_4's request would be blocked until the end of the shipment transmission to BDC_2. After BDC_4 has received the shipment, it stores it to satisfy further requests on the same shipment immediately (Store) and then sends it to *R* (step 9). *R* can now inform the user about the availability of new content in channel *C*.
The Minstrel broadcasting algorithm can be summarized as follows: A broadcaster *B* distributes reference information (samples) using MADP to the transport system. The transport system consists of hierarchical layers of BDCs that further distribute the samples to receivers. Regarding MADP, all BDC configurations actively disseminate samples. With respect to MRRP, a BDC can be viewed as a generalized cache. It can exist in three configurations:

- Repeater: a "pre-loaded cache" that immediately requests the shipments corresponding to the received samples; brings shipments "closer" to the receiver *R*; this causes slightly higher traffic but facilitates faster responses
- Cache: the usual caching functionality; can also be viewed as an "on-demand" repeater
- Proxy: facilitates access to channels where receivers cannot gain direct access

R can request a shipment corresponding to a received sample from its SA via MRRP. The request is propagated up the transport system—possibly until it reaches *B*—until it can be satisfied by a component, i.e., the shipment is transferred to *R*.
A detailed description of the broadcasting protocols is given in Section 4.7.

4.2.2 A generalized Picture

Section 4.2.1 has provided a slightly simplified view of Minstrel's broadcasting strategy: It considered only one information source, one channel, one broadcaster, and one receiver. To provide a complete picture, this section briefly gives a generalized view of broadcasting in Minstrel. Figure 4.3 shows a typical real-word configuration:

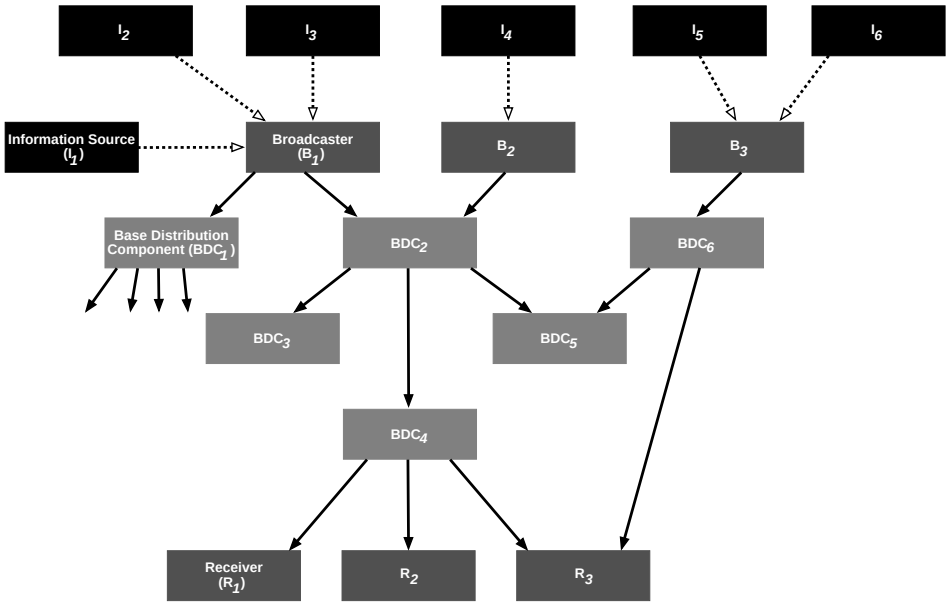

Figure 4.3: A typical broadcasting configuration

In the general case a broadcaster can have several information sources, which provide content that the broadcaster distributes via a number of channels. For example, B_1 is fed by I_1, I_2, and I_3. A broadcaster can feed several BDCs which in turn feed other BDCs or receivers. For example, B_2 feeds BDC_2, which in turn feeds BDC_3, BDC_4, and BDC_5. Receivers on the other hand can receive multiple channels from multiple SAs. For example, R_3 has two SAs—BDC_4 and BDC_6—and receives channels from them, which in turn means that it can receive channels from all broadcasters and thus content from all information sources shown in Figure 4.3.

A Minstrel broadcaster separates the concerns of channel handling from the information source. Likewise the transport system frees broadcasters from the issues of scalability and efficient distribution. An additional benefit of delegating the distribution to a separate transport system is that the transport system infrastructure can be reused. This means that broadcasters can use an existing distribution infrastructure and need not set up a new one. For example, BDC_2 and its downstream infrastructure are reused by B_1 and B_2. With this conceptual separation it is also possible that a transport infrastructure is run by a dedicated transport service provider, which sells this service to companies running broadcasters. The same applies to information sources and broadcasters.

In a small setting (e.g., inside the Intranet of a company) with a limited number of receivers, the transport infrastructure (BDCs) may not be necessary, and *B* could directly contact the receivers. This special case, however, is implicitly covered and is not considered further in the following descriptions.

The configuration shown in Figure 4.3 can be reduced to the setup depicted in Figure 4.2 without constraining generality. Thus Figure 4.2 will be used as basis for discussing the issues of Minstrel's broadcasting concepts in the following sections, since it simplifies the presentation and makes the concepts easier to understand.

4.3 Channel Subscription

From a conceptual point of view, receivers subscribe to channels that they want to receive from broadcasters. At the infrastructure level, however, they subscribe a channel at a service agent (SA). The same applies to BDCs. For example, in Figure 4.3, BDC_4 is the SA of R_3 for channels from B_1 and B_2, and BDC_6 is its SA for channels from B_3. Likewise, BDC_5 has two SAs: BDC_2 for channels from B_1 and B_2, and BDC_6 for channels from B_3. This concept of subscribing a channel not directly at its source but at a "closer" node (in terms of network properties) is Minstrel's approach to support scalability.

Generally speaking, a *service agent* (SA) is a broadcaster or BDC and denotes a node's (BDC or receiver) access point to the Minstrel infrastructure. All communication between a receiver or BDC and the Minstrel infrastructure is carried out via a SA (channel subscription, sample distribution, content request, etc.). The SA interacts with the Minstrel system on behalf of the receiver or BDC and vice versa. Each receiver or BDC has a small set of SAs and receives a specific channel from exactly one SA. SAs are *relative to a channel*. For example, if B_1 would broadcast the channels C_1–C_{10}, and B_3 would broadcast the channels C_{20}–C_{25}, R_3 would subscribe C_4 at BDC_4 and C_{21} at BDC_6.

To support scalability, a "close" node (in terms of network properties) that offers a channel of interest must be used as SA. This must be enforced by local configurations and organizational rules. BDCs and broadcasters must decide how many and what type of subscriptions (from BDCs and/or receivers) they accept. For example, BDC_2 may be configured to accept a maximum of 20 subscriptions from other BDCs but reject subscription requests from receivers. R_1 and R_2 on the other hand may be required by their channel service provider to use BDC_4 as their SA.

Each SA stores information about its subscribers in its subscription database. Subscriptions only have a local scope and are not forwarded to other parties by default. This design supports the highest possible level of privacy for the subscribers. Only their SA knows their private data, and subscribers can choose their SA depending on the level of privacy it guarantees.

However, it may be necessary or intended that data of the subscribers be made available to other parties. For example, to tailor the content of a channel to the users' needs, an SAs could send such data up the distribution hierarchy to the broadcaster. This is outside the scope of Minstrel and depends on the business models of the participating parties.

Minstrel's subscription model can be formalized in terms of a subscription relation and subscription distance. The *subscription relation* is defined as follows: An entity *A* is subscribed to an entity *B* for channel *C* (denoted as $\mathcal{S}(A, B, C)$), if

$$\mathcal{S}(A,B,C) \Leftrightarrow \mathcal{S}_{\mathcal{D}}(A,B,C) \vee \mathcal{S}_{\mathcal{I}}(A,B,C) \tag{4.1}$$

$$\mathcal{S}_{\mathcal{D}}(A,B,C) \Leftrightarrow A \text{ is } B\text{'s SA for channel } C \tag{4.2}$$

$$\mathcal{S}_{\mathcal{I}}(A,B,C) \Leftrightarrow \exists X : \mathcal{S}_{\mathcal{D}}(A,X,C) \wedge \mathcal{S}_{\mathcal{I}}(X,B,C) \tag{4.3}$$

This means that *A* is subscribed to *B* for channel *C*, either if *A* is *B*'s SA for *C*, i.e., (4.2) is true, or there exists a sequence of entities from *A* to *B* for which (4.2) is always true (4.3). (4.2) describes a *direct subscription* ($\mathcal{S}_{\mathcal{D}}$), whereas (4.3) denotes an *indirect subscription* ($\mathcal{S}_{\mathcal{I}}$). In Figure 4.3

$\mathcal{S}(B_1, BDC_2, C_4)$ (direct subscription) and $\mathcal{S}(B_1, BDC_4, C_4)$ (indirect subscription) would be true, whereas $\mathcal{S}(BDC_2, BDC_6, C_4)$ would be false.
The *subscription distance* $\Delta_{\mathcal{S}}(A, B, C)$ between two entities is defined as:

$$\Delta_{\mathcal{S}}(A,B,C) = \begin{cases} 1 & \Leftrightarrow \quad \mathcal{S}_{\mathcal{D}}(A,B,C) \\ 1 + \Delta_{\mathcal{S}}(X,B,C) & \Leftrightarrow \quad \mathcal{S}_{\mathcal{D}}(A,X,C) \wedge \mathcal{S}_{\mathcal{I}}(X,B,C) \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

According to (4.4), $\Delta_{\mathcal{S}} = 1$ for direct subscriptions and $\Delta_{\mathcal{S}} > 1$ for indirect subscriptions. In Figure 4.3, for example, $\Delta_{\mathcal{S}}(B_1, BDC_2, C_4)$ equals 1 and $\Delta_{\mathcal{S}}(B_1, BDC_4, C_4)$ equals 2.
Two entities A and B can communicate via MADP or MRRP only if $\Delta_{\mathcal{S}} = 1$:

$$MADP(A,B,C) \Leftrightarrow \Delta_{\mathcal{S}}(A,B,C) = 1 \tag{4.5}$$

$$MRRP(A,B,C) \Leftrightarrow \Delta_{\mathcal{S}}(A,B,C) = 1 \tag{4.6}$$

Thus the $\mathcal{S}$ and $\Delta_{\mathcal{S}}$ implicitly define the layers in the Minstrel infrastructure.

4.4 Data Structures used in Protocols

The Minstrel protocols use two key data structures—samples and shipments. Samples are distributed to recipients (BDCs, receivers) via MADP and those in turn request shipments via MRRP based on the information in the samples. This section describes the relevant details of these data structures.

4.4.1 Sample

A *Sample* models the commercial concept of *promotion through product samples*. To gain the customer's interest, a producer sends a sample of his goods to the customer together with an offer that defines the business terms under which he is willing to conclude a deal. If the customer is indeed interested by the promotion, s/he can order a shipment of the goods according to the defined business terms.
In the case of the Minstrel push system this is translated as follows: a broadcaster sends information about new content that is available in a channel to the channel's subscribers. This information consists of a product sample and the business terms and must be detailed enough to allow the recipient to decide whether to request a Shipment. For example, in a channel that offers high-quality images (several megabytes), a Sample would hold a rather small, very low-quality "thumbnail" of a newly available image as well as the information on its price, size, image format, etc. On the basis of this information, a recipient may decide to order a shipment of the high-quality image.
Figure 4.4 shows the UML [147] class diagram for a Sample.

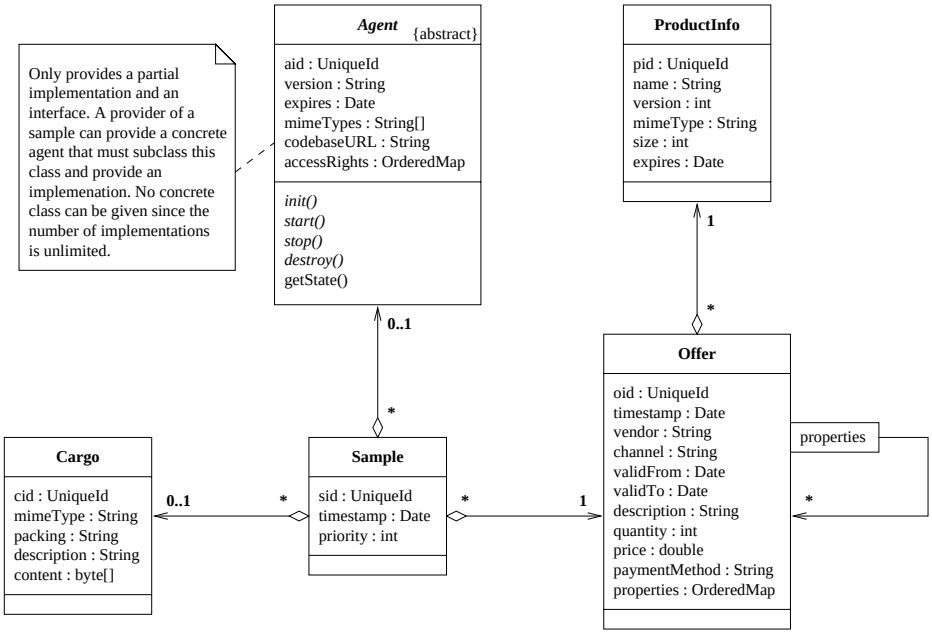

Figure 4.4: UML class diagram for a Sample

In terms of the implementation, a *Sample* is uniquely identified by a globally unique sample ID, is timestamped and priorized, and holds an *Offer*, some *Cargo*, and an *Agent*.

4.4.1.1 Offer

An *Offer* models the business terms. It has a globally unique ID which must be specified when ordering a Shipment: The requester must specify the ID of an Offer that defines the respective business conditions and indirectly identifies the "product" to be shipped. An Offer gives a short *description* of the "product," defines what *quantity* of the "product" is offered and its *price*.[2] If payment is necessary, the Offer defines the *paymentMethod* and possible parameters (which are not shown in Figure 4.4). For example, this could be a payment server's URL, that allows the recipient to commit a payment transaction for a Shipment it requests. Minstrel provides a generic payment model that supports various payment methods and business models, such as pay-per-view and flat fees. The current version of Minstrel includes an implementation of the Millicent [54] micro-payment protocol as payment method for a pay-per-view business model. A detailed description of e-commerce in the Minstrel system is given in Section 6.2.
Each Offer comes from a specific *vendor* and may only be valid for a certain period which is defined by the *validFrom* and *validTo* fields. Offers (and thus Samples) are always relative to a *channel*. If a product is to be announced via multiple channels a dedicated Offer for every channel is required. The channel identifier allows all components of the Minstrel system to relate Samples (via the contained Offer) to channels. This is important since every Minstrel component

[2] A currency for the price can also be specified, but is not shown in Figure 4.4.

(BDCs, receivers) that receives Samples has only one interface for receiving them (a dedicated receiving interface for every channel would be inefficient). The division into channels is done after the reception of a Sample and thus requires such an identifier. The channel names are textual and human-readable to make them meaningful to users and easier to understand. Nevertheless, channel names must be unique which is provided by a simple hierarchical naming scheme. If Minstrel is successful, it is planned to apply for a global namespace for channel names at the Internet Assigned Numbers Authority (IANA).
To relate an Offer to other objects (other Offers, Samples, etc.), every Offer can hold a list of name-value pairs in its *properties*. For example, an Offer could announce the availability of some Java software contained in a single JAR file (Java Archive), and this JAR file could replace older versions of software, say three other JAR files. Then this Offer could specify the following properties (pseudo notation): `(("replaces", (oid1, oid2, oid3)))`. A list of keywords and their semantics is predefined in Minstrel. However, this list can easily be extended and tailored to the needs of particular applications.
Each Offer is related to a specific product that is described in the *ProductInfo* of the Offer. The product is identified by a globally unique ID. Its *name*, *version* information, and expiry date are given as well. The *mimeType* and *size* fields provide criteria for determining whether to request a Shipment. For example, a user configuration could require to request only `image/jpeg` of a size less than 100kB. Additional criteria for the request decision are provided by the Sample's *Cargo*.

4.4.1.2 Cargo

The *Cargo* object of a Sample holds the actual product sample (e.g., a low-quality "thumbnail" of a high-quality image). Generally speaking, Cargo objects hold "content", whatever this may be: a "thumbnail" (Sample), a small HTML document (Sample), a high-quality image (Shipment), a JAR file (Shipment), or a pushlet (Shipment). Since Minstrel does not constrain the size of Samples (although it can be configured to accept only Samples up to a maximum size), the Cargo and its size must be chosen very carefully. Too large Samples make the dissemination protocols slow and inefficient; too little information may not support the recipient in his/her decision to request a Shipment or drop the Sample. As a rule of thumb, Samples should not exceed a size of 5–10kB.
Some channels can have empty Cargo fields in their Offers: For example, a channel for software distribution would require only a short human-readable description of the software to be distributed. Another special case is the distribution of small amounts of information. If the information is small enough, for example, some stock quotes that require only a short textual message that easily fits into a Sample, Minstrel can be configured to include this content—which actually is the shipment—into the Sample and its Cargo. This concept can further cut down on network bandwidth consumption, since it makes a followup-Shipment superfluous. However, this must be stated explicitly in the Sample and is not always possible or desirable. For example, in the case of a pay-channel the recipient gets the "product" before it was payed. This tradeoff should be considered carefully before setting up such a configuration.
To make Cargo a general-purpose container, *content* is defined to be a sequence of bytes—the

most general data type. Its remaining fields—*mimeType* and *packing*—provide information that is necessary for the decoding and processing of Cargo's *content*.

4.4.1.3 Agent

Minstrel includes content handlers for the most frequently used MIME [46, 47, 48, 49, 114] types on the Internet. Some Cargo objects, however, may hold content types that cannot be handled by Minstrel's standard handlers. In these cases the provider of the Cargo can supply a specialized *Agent* to process the *content* of the Cargo object (if the Cargo, however, holds a MIME type that Minstrel can process, no Agent needs to be supplied with the Sample). The agent concept can also be exploited for other purposes.
Again, an Agent is identified by a globally unique ID, holds *version* information, and an expiry date. It defines a list of *mimeTypes* it can handle and the *accessRights* it requires on the user's machine (a discussion of security issues related to mobile code in Minstrel is given in Section 6.1.3).
Agent is an *abstract* class. Abstract classes cannot be instantiated, as they only provide a partial implementation. They can specify a set of data fields, some methods that they implement, and an interface that derived classes must provide in their implementations [53]. A concrete Agent provided by a supplier of a Sample—and sent as part of a Sample object—must subclass Agent and provide implementations for the *init(), start(), stop(),* and *destroy()* methods (note that the *getState()* method is not abstract; however, it can also be redefined by subclasses of Agent). This design was chosen to provide a uniform interface to agents that Minstrel can rely on. *init()* is called directly after instantiation of the Agent, *start()* to actually start its execution, *stop()* to stop the execution of the Agent, and *destroy()* to dispose the Agent, and free all the resources it consumed (memory, connections, files, etc.).
getState() is used by Minstrel to request information on the execution state of the Agent. For example, if the Agent performs a computation-intensive task, Minstrel must be able to check the progress of this task or whether an Agent has terminated after a call to its *stop()* method. Agents execute as separate threads. In Java 2—the implementation platform of Minstrel—stopping of threads via their *stop()* method is no longer possible. Instead, the recommended method is to have the thread repeatedly check a variable whether it should terminate [161]. An Agent's implementation must follow this rule: *stop()* should be used to indicate Minstrel's request for termination, and upon termination the Agent should make its termination status available via *getState()*.
Since Samples are required to be rather small Agents can have only a very limited size. Thus Minstrel uses a two-level design. If the Agent that comes with a Sample is small enough, it comes with all its required code and data. This, however, will seldom be the case. If the Agent object exceeds a certain size, it comes as a *light-weight* Agent. It only contains the minimal mandatory code for the required methods and a minimal amount of data; the code providing the full functionality of the Agent resides at the location specified by the *codebaseURL* field. When the Agent is instantiated, Minstrel creates a dedicated *classloader* [55] for the Agent that allows it to dynamically load required code on demand. This implementation technique is comparable to standard bootstrapping techniques. For further optimization, Agents that have already been used, i.e., have been loaded in the way described above, can be stored locally. Since Agents are

uniquely identified by their attributes (*aid*, *version*, *expires*, etc.) consecutive requests for the same Agent can be satisfied by this local copy. This further cuts down on the transmission costs and delays and allows to dynamically extend the functionality of the receiver software.

4.4.2 Shipment

The counterpart of a Sample is a *Shipment* that is requested if the data provided in the Sample sparks a positive decision to issue such a request. Figure 4.5 shows the UML [147] class diagram for a Shipment.

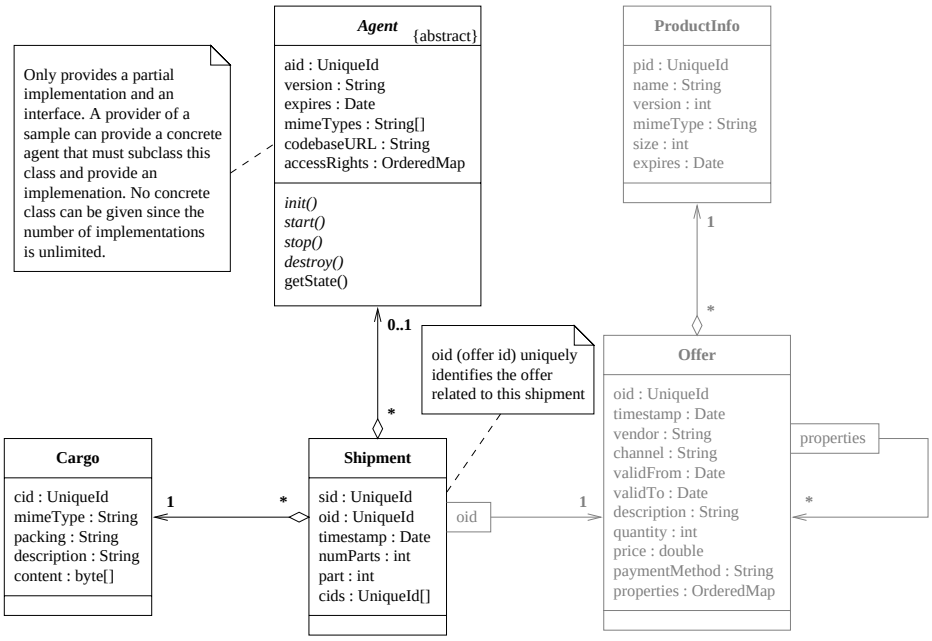

Figure 4.5: UML class diagram for a Shipment

As a Sample, any Shipment is timestamped and identified by a unique ID (*sid*). It carries the unique ID of the Offer (*oid*) it corresponds to. This ID was given as a parameter in the MRRP request the client issued to obtain the Shipment. It can be used to relate the Shipment to the Offer that defines the business terms and which already resides at the client (having been received with the corresponding Sample). The ProductInfo for this Shipment can be accessed indirectly via the Offer. Offer and ProductInfo are included in Figure 4.5 to show explicitly how a Shipment relates to an Offer. However, since they are *not* part of a Shipment (this would require an inefficient retransmission), they are shown in gray.

As already mentioned, Shipments can be rather large—up to several megabytes. For an efficient transmission and to make the transmission process more fault-tolerant, a large Shipment can be split into several smaller ones. *numparts* defines the total number of parts and *cids* holds a list of all the Cargo IDs the full Shipment consists of. *part* defines the sequence number of the Cargo contained in the current Shipment in relation to the other parts of the complete shipment. If a

Shipment has been divided into several partial Shipments, the receiver waits until it has received all parts before it notifies the user.
The Cargo of a Shipment holds the actual content. If necessary an Agent to process the Cargo can be included in the Shipment. The Cargo and Agent objects of a Shipment are identical to those described above for a Sample. However, it is important to note that a Sample *can* include a Cargo object while a Shipment *must* include one for obvious reasons.

4.5 Processing of Shipments

After a Shipment has been received the user is notified and the Shipment can be accessed. In the case of a multi-part Shipment, all parts must have been received before notifying the user. Prior to processing, a multi-part Shipment is assembled into a single-part Shipment.
The processing of a Shipment works as depicted in Figure 4.6.

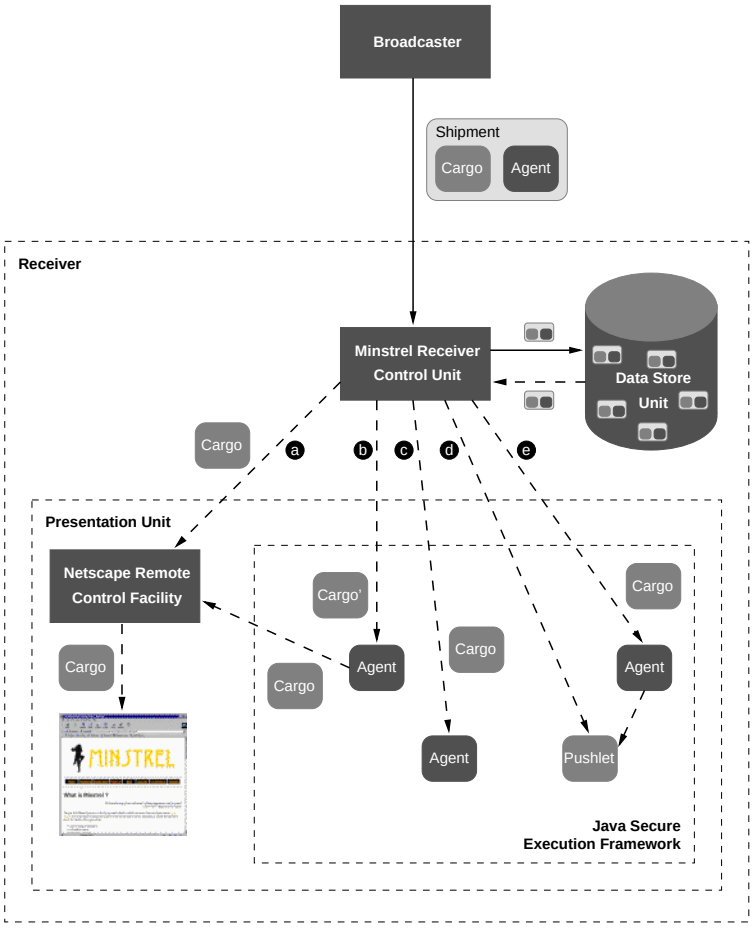

Figure 4.6: Processing of Shipments

The Minstrel Receiver receives a Shipment containing a Cargo and possibly an Agent. After

the reception is completed by the protocol components of the Receiver, the *Minstrel Receiver Control Unit* (MRCU) is notified. The MRCU (see Section 5.1.1) is the central component of the Receiver. It directly or indirectly controls and manages all other Receiver components. It controls the *Data Store Unit* (DSU) and the *Presentation Unit* of the Receiver and interacts with the transport system via MADP and MRRP. The user can interact with the MRCU via a graphical user interface.

The first step taken by the MRCU is to store the newly arrived Shipment in the DSU according to the information contained and referenced in the Shipment. This means that the MRCU retrieves the Offer corresponding to the Shipment from the DSU and stores the Shipment in the DSU according to the channel defined in the Offer. The DSU (see Section 5.1.3) is a persistent storage that can store all Minstrel data objects (Sample, Shipment, Offer) and provides flexible and fast retrieval methods.

After the Shipment has been stored, the user is notified of the newly available content. If the user decides to view the newly arrived Shipment, the MRCU retrieves the Shipment from the DSU. If the Shipment's Cargo contains content of a MIME type known to *Netscape Communicator* [125] and which requires no further processing, it is given to the *Netscape Remote Control Facility* (NRCF) of the Presentation Unit which instructs Netscape Communicator to display it (case (a) in Figure 4.6). If feasible links inside the Cargo are available and Netscape Communicator uses the Receiver as its proxy, then the user can also interact with the Cargo, for example, by requesting further shipments referenced in it.

Netscape Communicator (or its free version Mozilla [115]) was included into the Presentation Unit since it supports a wide range of MIME types.[3] This greatly simplifies the implementation of the Receiver, since it can rely on the capabilities of Netscape Communicator for displaying most existing MIME types. Moreover, support for new MIME types is likely to be included quickly into Netscape Communicator.

The integration of Netscape Communicator (Mozilla) is done by exploiting its remote control interfaces. The Netscape Remote Control Facility (NRCF) relies on these interfaces and offers services to conveniently control the operation of Netscape Communicator (Mozilla). NRCF works with an out-of-the-box version of Netscape Communicator (Mozilla) and the *Java Plug-in* [159] provided by Sun Microsystems. A description of the NRCF is given in Section 5.1.2.

In the case that the MIME type is not supported, the Shipment must include an Agent which can process the Cargo. In this case the Agent is instantiated, started, and provided with the Cargo for further processing. If the Agent converts the Cargo into a MIME type known to Netscape Communicator, the converted Cargo is given to the NRCF for displaying it (case (b) in Figure 4.6). For example, the Cargo could hold a GIF image which was compressed using an unsupported compression scheme. The Agent uncompresses the GIF and then hands the GIF to the NRCF. However, it may also be possible that only the Agent can process and display the Cargo (case (c) in Figure 4.6).

If the Cargo holds a pushlet (case (d) in Figure 4.6), it is instantiated and started. To protect the Receiver from malicious code all agents and pushlets are executed inside Minstrel's *Java*

[3] Microsoft's Internet Explorer was also evaluated for this purpose. Netscape Communicator was given the preference, however, since it is also available for non-Microsoft platforms.

Secure Execution Framework (see Section 6.1.3). In the case that the pushlet comes in a special format, it may also be necessary to process it with an Agent before it can be executed (case (e) in Figure 4.6).
Descriptions of the Receiver and its main components—MRCU, DSU, and NRCF—are given in Section 5.1. JSEF is described in Section 6.1.3.

4.6 Discussion of the Broadcasting Strategy

Minstrel's broadcasting strategy was designed to support efficient and timely information dissemination while providing a high degree of scalability. The scalability issues Minstrel needed to address were:

- Server load
- Network bandwidth consumption
- Delay.

Minstrel uses a hybrid broadcasting strategy on top of two protocols: MADP to actively disseminate samples of small size and MRRP for requesting large shipments.
This section gives a detailed discussion of Minstrel's broadcasting strategy.

4.6.1 Design Issues of the Broadcasting Strategy

The most fundamental question is, why a hybrid broadcasting scheme was chosen. The answer is straightforward and can be deduced directly from the design goals of Minstrel. The broadcasting process was designed to be as efficient and scalable as possible while not relying on special multicasting infrastructures. To reach this goal only small amounts of information (samples) are distributed in an iterative process to a small number of direct recipients in an adjacent lower layer. This facilitates low bandwidth consumption due to the small size of samples (some kilobytes) and the limited number of recipients. To support high numbers of recipients this scheme is replicated in hierarchical layers in a tree structure, which also distributes the broadcasting load among the disseminating components. Recipients are informed quickly and efficiently via samples while network bandwidth is conserved, since the much larger shipments (up to several megabytes) are requested only if a recipient really wants them. This means that shipments which are not explicitly requested are not disseminated. For faster availability, repeaters automatically mirror shipments and bring them closer to the final recipients.
The additional latency introduced by Minstrel's transport system is outweighed by its excellent scalability. As long as data need not be disseminated in real-time, this latency is acceptable because it is rather small and the transport system has only a small number of levels. Minstrel addresses only non-real-time information distribution, since most information to be disseminated over the Internet will be of that type. For real-time data dissemination, highly optimized protocols are necessary that are beyond Minstrel's scope.

4.6.2 Analysis of a concrete Scenario

The key scalability issue in Minstrel's hybrid broadcasting strategy is the efficient distribution of samples via MADP (push part). While pulling protocols such as HTTP [11, 42] and MRRP are well analyzed and have proven their scalability, only few analyses exist for push protocols such as MADP. This section analyzes the push part of Minstrel's broadcasting strategy on the basis of the concrete scenario shown in Figure 4.7:

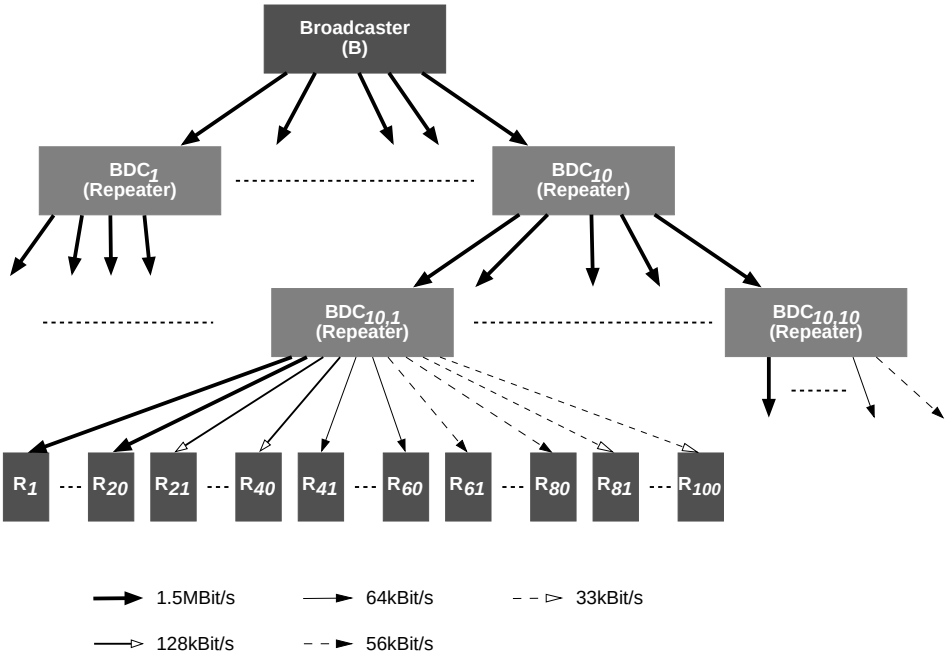

Figure 4.7: A concrete MADP scenario

- A broadcaster distributes samples via MADP to 10 first-level repeaters.
- Each first-level repeater feeds 10 second-level repeaters.
- Each second-level repeater feeds 100 receivers.
- The broadcaster and the first-level repeaters are connected via T1 (1.5MBit/s = 192kB/s) links.
- The first-level and the second-level repeaters are connected via T1 (1.5MBit/s = 192kB/s) links.
- The receivers are connected to the second-level repeaters via 33kBit/s (= 4.125kB/s) and 56kBit/s (= 7kB/s) modem lines, 64kBit/s (= 8kB/s) and 128kBit/s (= 16kB/s) ISDN lines, or T1 (1.5MBit/s = 192kB/s) links.

- Each second-level repeater serves 20% 33kBit/s receivers, 20% 56kBit/s receivers, 20% 64kBit/s receivers, 20% 128kBit/s receivers, and 20% T1 receivers.

The total number of receivers $receivers_{total}$ that can be reached in this scenario can be computed as follows:

$$\begin{aligned} receivers_{total} &= |BDC_i| * |BDC_{i,j}| * |R_k| \\ &= 10 * 10 * 100 \\ &= 10000 \end{aligned} \tag{4.7}$$

Let us assume the broadcaster obtains a weather satellite GIF image as new channel content from its information source and constructs a sample to disseminate via MADP. Assuming a typical sample size of 3.5kB[4] the relative bandwidth consumption and maximum sample transfer rates for the network connections of Figure 4.7 are:

Bandwidth	Bandwidth/Sample	Samples/Minute
1.5MBit/s	0.23%	3291
128kBit/s	2.73%	274
64kBit/s	5.47%	137
56kBit/s	6.25%	120
33kBit/s	10.61%	70

Table 4.1: Bandwidth consumption and sample transfer rates

Sending a 3.5kB sample to all repeaters in the scenario of Figure 4.7 requires the following network bandwidth in the transport system ($B \rightarrow BDC_i + BDC_i \rightarrow BDC_{i,j}$):

$$\begin{aligned} bandwidth_{transport} &= \Big(|BDC_i| + |BDC_{i,j}|\Big) * 3.5 \\ &= (10 + 100) * 3.5 \\ &= 385\,[kB] \end{aligned} \tag{4.8}$$

Sending this sample to all receivers then requires a total network bandwidth of:

$$\begin{aligned} bandwidth_{total} &= bandwidth_{transport} + receivers_{total} * 3.5 \\ &= 385 + 10000 * 3.5 \\ &= 35385\,[kB] \end{aligned} \tag{4.9}$$

[4]This number is based on tests.

Without the transport system 385kB less would have been needed. This overhead, however, is small compared to the huge delay that would have been introduced by sending the sample sequentially to every receiver (see Section 4.6.3). In the sequential case, the computing and network resources of only a single server would be used and all the costs of sending 35MB would be charged to the authority running this server. It must be kept in mind that these are the costs of a *single* sample transmission.
With a transport system as given in Figure 4.7, the enormous bandwidth of 35MB is divided up between *B* (35kB), the BDC_i (35kB each), and the $BDC_{i,j}$ (350kB each). The consumption of computing resources is distributed alike and the delay also is significantly lower than in the case of sequential distribution.

4.6.3 MADP Worst-case Delay

Without constraining generality, let us further assume that broadcasters and repeaters order their subscribers according to the network bandwidth of the connection to the subscriber. Thus the subscriber with the fastest connection would be the first to receive a sample. If two subscribers have an equally fast network link, then the one with the lower ordinal number ist chosen first.
Then the maximum delay in the scenario of Figure 4.7 occurs for the receiver which is last to be fed by its repeater and every repeater on the path to the receiver is last to get the sample from its feeding repeater (or broadcaster). Under this assumption, $R_{BDC_{10,10},100}$ would then be the receiver with the maximum delay in Figure 4.7.
Let $delay(source, destination, size)$ denote the time necessary to transfer $size$ kB from node $source$ to node $destination$. The worst-case (maximum) delay for the scenario of Figure 4.7 can then be calculated as:

$$
\begin{aligned}
delay(B, R_{BDC_{10,10},100}, 3.5) &= \sum_{i=1}^{10} delay(B, BDC_i, 3.5) + \sum_{j=1}^{10} delay(B_{10}, BDC_{10,j}, 3.5) \\
&\quad + 20 * \Big(delay(BDC_{10,10}, R_{BDC_{10,10},1}, 3.5) \\
&\qquad + delay(BDC_{10,10}, R_{BDC_{10,10},21}, 3.5) \\
&\qquad + delay(BDC_{10,10}, R_{BDC_{10,10},41}, 3.5) \\
&\qquad + delay(BDC_{10,10}, R_{BDC_{10,10},61}, 3.5) \\
&\qquad + delay(BDC_{10,10}, R_{BDC_{10,10},81}, 3.5) \Big) \qquad (4.10) \\
&= 10 * \frac{3.5}{192} + 10 * \frac{3.5}{192} \\
&\quad + 20 * \left(\frac{3.5}{192} + \frac{3.5}{16} + \frac{3.5}{8} + \frac{3.5}{7} + \frac{3.5}{4.125} \right) \\
&= 20 * 0.018229167 \\
&\quad + 20 * (0.018229167 + 0.21875 + 0.4375 + 0.5 + 0.\overline{84}) \\
&= 40.823864\,[s]
\end{aligned}
$$

This also means that all receivers could be notified within 1 minute. Figure 4.8 shows the delays for every receiver in Figure 4.7.

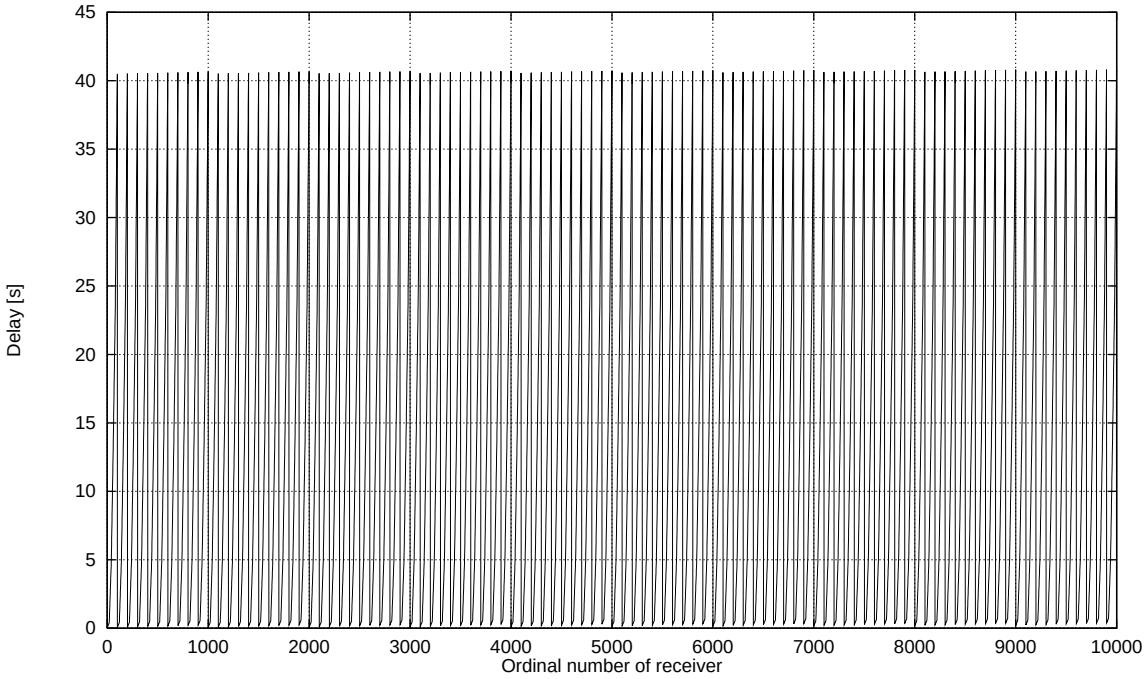

Figure 4.8: Minstrel distribution delays

This figure should be interpreted as follows: The distribution delay is approximately equally distributed over the 100 $BDC_{i,j}$ which are shown by the 100 "spikes" in Figure 4.8. The spikes start slightly over 0, which is due to the distribution delays introduced by sending the sample from *B* to the 10 BDC_i and then propagating this sample to the 100 $BDC_{i,j}$. Since B, the BDC_i, and the $BDC_{i,j}$ are connected with fast T1 links and the sample is rather small, this causes only very small delays, ranging from 0.036458333 seconds for $BDC_{1,1}$ to 0.36458333 seconds for $BDC_{10,10}$. Because these delays are negligible compared to the delays of sending the sample from the $BDC_{i,j}$ to the receivers, they are hard to identify in Figure 4.8.
Every $BDC_{i,j}$ sends the sample sequentially to 100 receivers. Thus the delay depends on the ordinal number of the receiver relative to the $BDC_{i,j}$ it is connected to. The delay for every receiver R_i must take into account the delays of the receivers R_1–R_{i-1} connected to the same BDC. Since the receivers connected to different BDCs are fed in parallel, the delays of these receivers only differ according the delays introduced by sending the sample to the relevant BDCs. For example, $delay(B, R_{BDC_{1,1},1}, 3.5)$ and $delay(B, R_{BDC_{1,2},1}, 3.5)$, only differ in the additional delay caused by sending the sample to $BDC_{1,2}$, which is 0.018229167 seconds higher than the delay for $BDC_{1,1}$. The same applies to all BDCs and receivers and also explains the parallel "spikes" and the direct proportionality of the delay to the ordinal number (relative to the feeding BDC). However, the upper limit for all delays is 40.823864 seconds.
Each individual "spike" in Figure 4.8 resembles the one shown in Figure 4.9.

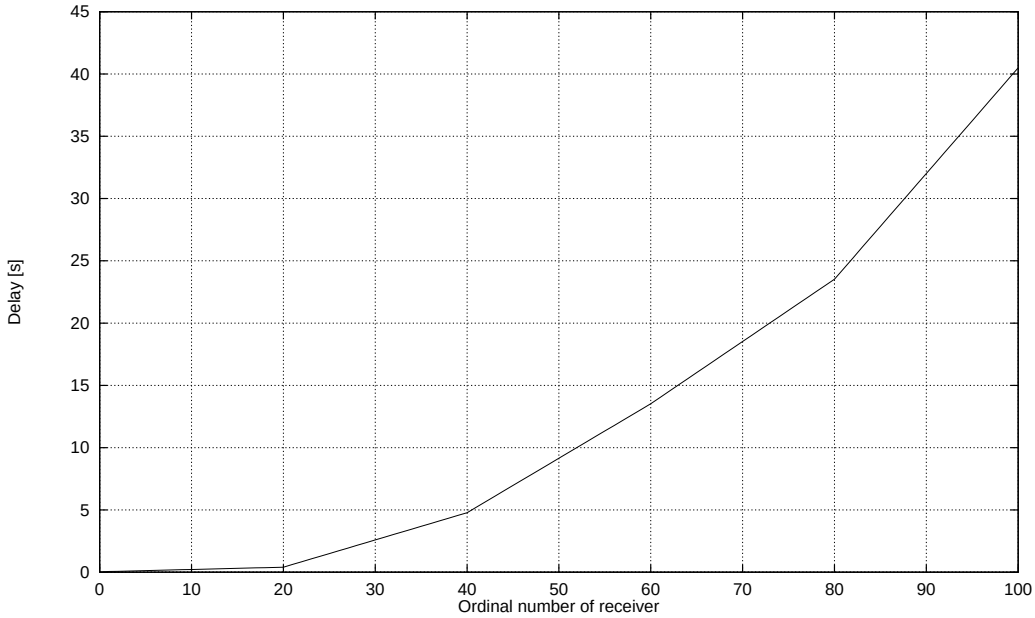

Figure 4.9: Minstrel distribution delay for sequential distribution by 1 BDC

The distribution of the delays as depicted in Figure 4.10 shows that most receivers will get the sample rather fast and that only 100 receivers are in the the interval with the maximum delay.

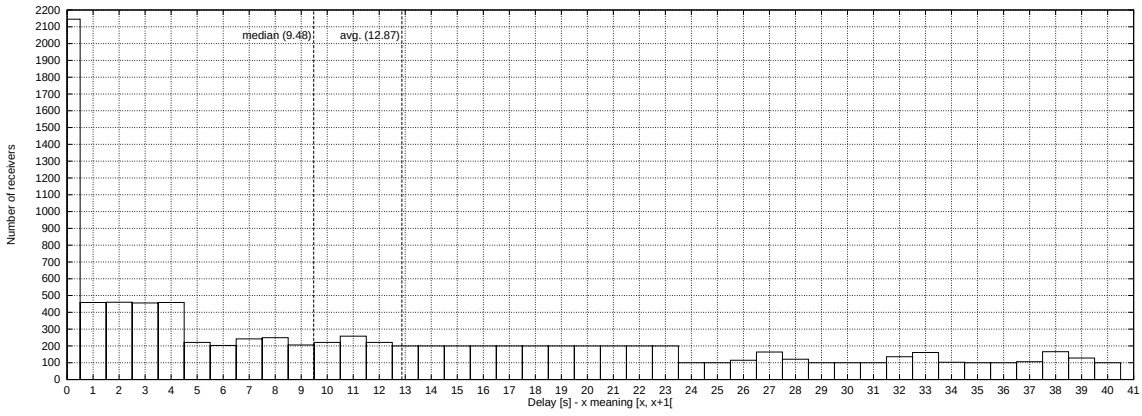

Figure 4.10: Distribution of delays in Minstrel

The average delay is 12.87 seconds and the median (50% of the delays higher and 50% lower) is 9.48 seconds. 72% of the receivers have a delay of less than 20 seconds (50% of the worst-case delay) and approximately 58% have a delay that is less than the average delay.
The maximum delay for distributing a 3.5kB sample serially to 10,000 receivers with the same links for the receivers as in Figure 4.7 and the receivers being sorted according to their network connection to *B* (fastest first) can be computed as:

$$
\begin{aligned}
delay(B, R_{10000}, 3.5) &= \sum_{i=1}^{10000} delay(B, R_i, 3.5) \\
&= 2000 * \Big(delay(B, R_1, 3.5) + delay(B, R_{2001}, 3.5) \\
&\qquad + delay(B, R_{4001}, 3.5) + delay(B, R_{6001}, 3.5) \\
&\qquad + delay(B, R_{8001}, 3.5) \Big) \qquad (4.11) \\
&= 2000 * \left(\frac{3.5}{192} + \frac{3.5}{16} + \frac{3.5}{8} + \frac{3.5}{7} + \frac{3.5}{4.125} \right) \\
&= 2000 * (0.018229167 + 0.21875 + 0.4375 + 0.5 + 0.\overline{84}) \\
&= 4045.928\,[s]
\end{aligned}
$$

This means that the last receiver gets the sample with a delay of 4045.928 seconds (1 hour 7 minutes 25.928 seconds). It is considerably higher than the maximum delay of the Minstrel distribution algorithm (besides the very inefficient resource utilization of one machine and its network connection). Minstrel has only 1.0090111% of the serial distribution delay in the worst case. Serial distribution, however, is still used, for example, by mailing lists (see Section 3.2.1). Figure 4.11 compares the delays for serial distribution with the delays of Minstrel.

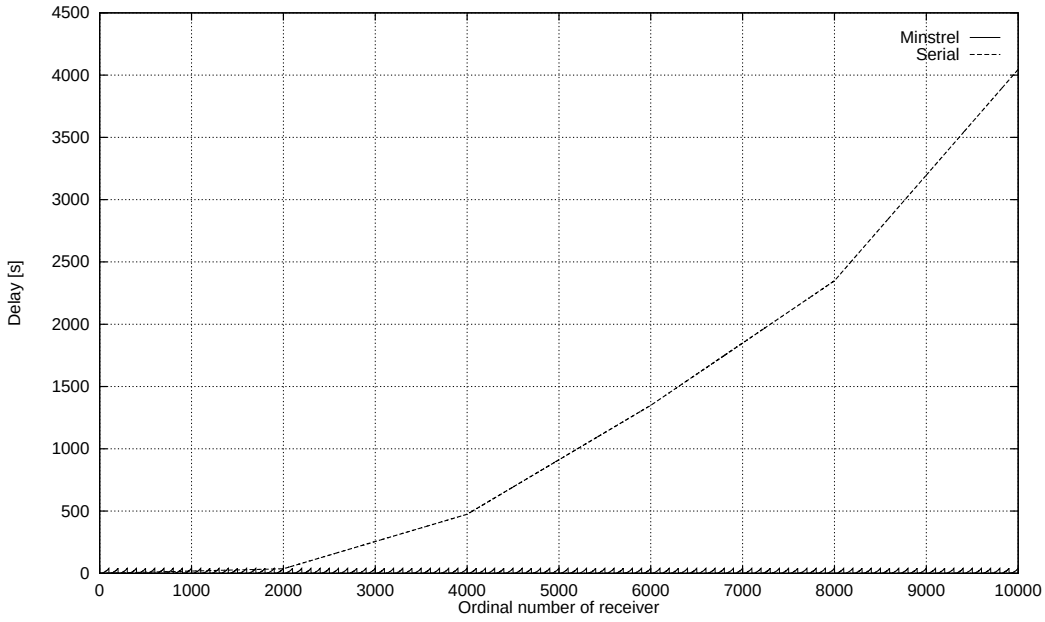

Figure 4.11: Serial distribution delays vs. Minstrel distribution delays

As stated before the figures above provide only an approximation of the actual system behavior and take into account only bandwidth. They do not take into account other factors such as processing load on the broadcaster, repeaters and receivers, delays introduced by processing

of received information and dispatching to recipients, or queuing delays. However, this model seems justified, since bandwidth is still the most limiting resource. Bandwidth is so predominating in this setting that the other factors have only minor influence (if some other resource had a bigger impact this could be compensated by a faster machine, more memory, etc.). Another implicit assumption is that all components are online all the time, so that queuing delays need not be considered.

4.6.4 Collocation of Repeaters with Internet Service Providers

The rather fast T1 links between the broadcaster and the repeaters in Figure 4.7 have only minor influence on the distribution delays of Minstrel. If the broadcaster and the repeaters were connected via 128kBit/s links, then the maximum delay would only increase by 9.82% to 44.83 seconds.
Moreover, the assumption of T1 links is justified. The broadcaster and the repeaters would normally be set up at sites with fast network connections. A reasonable choice would be Internet Service Providers (ISPs) that are needed in any case by every site connected to the Internet. ISPs typically have high-speed connections to the Internet and other ISPs. By this collocation strategy a reasonably fast push-backbone would be available by organizational measures on top of an existing infrastructure.

4.6.5 MRRP and implicit Caching

So far only efficient sample distribution has been considered. The sample distribution process is done via the Minstrel Active Distribution Protocol (MADP). If the receiver decides to request the actual data (the shipment) described by the sample, the shipment is requested and transferred via the Minstrel Receiver Request Protocol (MRRP). Shipments can be rather large, up to several megabytes. Therefore efficiency, meaning low network bandwidth consumption and fast response time, is an important issue here as well.
Due to the structure of the Minstrel transport system MRRP efficiency comes at rather low cost. Receivers can send their MRRP requests only to one of their SAs. The SA can either satisfy the request, if it already has the shipment, or it requests the shipment from its SA. Since most of the SAs will be BDCs which are configured either as caches or repeaters (that also do caching) this builds up an implicit caching infrastructure that satisfies the requirements of low bandwidth consumption and reasonably fast response time. In contrast to systems like the WWW or Usenet news, where caching had to be introduced as a second-level concept to ensure scalability and efficiency, Minstrel already includes caching as a first-level concept.

4.7 Broadcasting Protocols

This section gives a detailed presentation of Minstrel's MADP and MRRP protocols. The protocols are described using UML state charts [147]. State charts were favored over other protocol notation techniques such as Petri nets because they support easy comprehension. However, they are not a formal definition that can be directly mapped into an implementation, but suffice for our purpose.
Since the state charts involve some parallelism, an additional notation convention was used to support readability of the diagrams: Actions that generate events and trigger state transitions in other (parallel) sections of the state charts are shown as white text on black background; the same notation is used for the according events.
Both MADP and MRRP are defined as a combination of two state charts: one for the receiving side and one for the sending side. The terms receiver and broadcaster are used in the following to indicate the communication role of an entity and do not necessarily denote broadcaster and receiver components. By convention every entity can have a receiver and a broadcaster role. For example, a BDC is an MADP receiver towards its SA and an MADP broadcaster towards the entities it feeds. It is also an MRRP broadcaster since it is an SA and thus receives shipment requests that it can either satisfy immediately or by sending an MRRP request to its SA, which means that it takes the role of an MRRP receiver. Table 4.2 lists the possible roles of entities.

Entity	Role	
	broadcaster	receiver
Broadcaster	MADP, MRRP	
Receiver		MADP, MRRP
BDC	MADP, MRRP	MADP, MRRP

Table 4.2: Roles of entities

MADP and MRRP are both based on Java's Remote Method Invocation (RMI) [66] paradigm. Thus all communication between Minstrel components (broadcaster, BDC, receiver) in the protocols is done via remote method calls.

4.7.1 Design Issues of the Protocols

In the previous descriptions of the broadcasting strategy it was implicitly assumed that all components are permanently connected to the Internet and online. This assumption, however, does not hold. For example, network disconnects can occur and many users access the Internet via modem dial-in. Thus MADP and MRRP must account for such situations. For MRRP the solution is straightforward since it is initiated by receiving party: Requests for shipments are delayed until network connectivity is available again or the requests initiate a dial-in process.
For MADP the problem is more complicated. If a subscriber cannot be reached by a distributing component (broadcaster or BDC), the relevant sample must be retained (queued) until the recipi-

ent becomes available again. The problems regarding the queuing strategy are similar to the ones faced by Internet mail transfer agents (MTAs) such as *sendmail* [21]. Some of the questions to be solved by any queuing system are, how long to retain a request, when to check whether the recipient is available, and the maximum amount of resources allocated to one recipient (number of samples, total size of samples, etc.). Minstrel supports a user-configurable queuing strategy to address these issues.
Content expiry is another problem to be considered in MADP and MRRP. Both samples and shipments have a certain lifetime after which they are no longer valid and are discarded. In the case of MADP, the distribution of a sample to a recipient may fail if the sample expires during the distribution process. This may be reasonable, if a recipient is not reachable for a long time. In other cases, for example, if samples expire for a certain recipient, despite the recipient's availability, measures may be necessary: The expiry period of samples can be increased; the network connection may be upgraded; a lower sample transmission rate can be tried, etc. For MRRP the situation is simpler. The expiry problem only occurs upon request of a shipment which has expired. In this case the requester is notified and no additional measures are required. It is important to note that for the reasons presented above, the delivery of samples and shipments cannot be guaranteed.
For reasons of simplicity, but without constraining generality, the following protocol descriptions show a non-parallel picture of the protocols, apart from some parallelism inside the protocol automatons. To ensure optimized protocol behavior, implementations must be highly parallel, however. For example, in the case of the MADP broadcaster, a set of concurrent worker threads (a so-called thread pool [91]) accesses a global pool of distribution jobs. The same concept is implemented in the MRRP receiver, where worker threads (also organized as a thread pool) access a global pool of retrieval requests to support efficient, parallel shipment retrieval. The receiver side of MADP on the other hand also applies multi-threading, otherwise the receiver could receive only one sample at a time while all other sample transmissions would be blocked. Also the broadcaster side of MRRP applies multi-threading, otherwise an SA could only satisfy one request at a time while all others would be blocked.
Multicast infrastructures or other specialized distribution substrates are not considered explicitly, since such infrastructures can be used without having to change the basic algorithms.

4.7.2 Minstrel Active Distribution Protocol

Figure 4.12 shows the UML state chart for the broadcaster side of the Minstrel Active Distribution Protocol (MADP).
The broadcaster side of MADP has two main threads of execution: the sender, which is responsible for distributing the samples to the subscribers, and the queue, which is in charge of queuing undeliverable samples, when the recipients cannot be reached.
As soon as a new sample becomes available, a list of tentative recipients is generated based on the subscription database. This list holds all local subscribers of the channel the sample belongs to. If this list is non-empty, filters are applied to determine the recipients that actually will get the sample. This filtering process serves two purposes. First, it is used to enforce protocol constraints such as the expiration of samples (expired samples need not be distributed). Second,

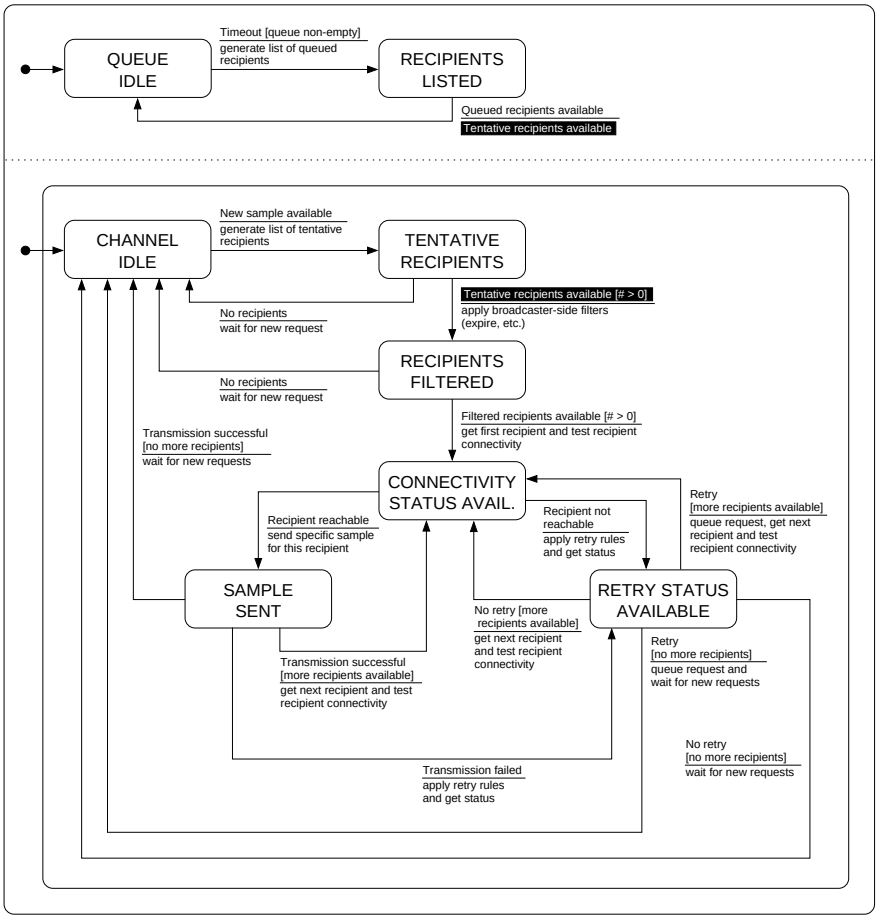

Figure 4.12: Minstrel Active Distribution Protocol (MADP) – Broadcaster side

every MADP broadcaster can apply application level filters to enforce its local distribution policy. For example, some sites may want to filter out advertisements or content not suitable for children (on the basis of subscriber information, content of the sample, policy rules, etc.).
Once the filters have been applied the final list of recipients is available and the distribution process can start. The first recipient is taken from the list and its connectivity status is checked. If it is reachable, then the sample is sent to it. If the transmission is successful, this procedure is repeated for the next recipient until the list is empty and the protocol engine goes into idle mode, waiting for new samples to distribute.
This distribution process may fail for various reasons. The two main cases of error are when:

- the connectivity test for the recipient fails, or
- the connectivity test for the recipient succeeds but the transmission of the sample fails.

In any case of failure the retry rules are applied. These rules take into account the local retry policy and configuration settings such as the maximum number of retries, the maximum num-

ber of samples queued for a specific receiver, the maximum total size of queued samples for a specific receiver, the maximum queue length, the maximum size of the queue, and the system load. Depending on the result, the job ("send sample *x* to receiver *y*") is either queued for retry or abandoned. If there are more recipients to be processed the distribution process continues. Otherwise the protocol engine goes into idle mode and waits for new samples to distribute.
The MADP queue operates in parallel to the distribution process described above. Its operation is based on regular intervals: A timeout occurs after a configurable amount of time and the queue generates a list of tentative recipients. This list does not necessarily include all queued recipients but can contain only a subset which is determined by protocol constraints, such as expiry date and priority of the sample, and the queue's configuration, such as channel-specific retry settings. When the list is available, an event is generated that triggers a distribution attempt by the sender component.
Figure 4.13 shows the UML state chart for the receiver side of MADP, which is rather simple compared to the broadcaster side.

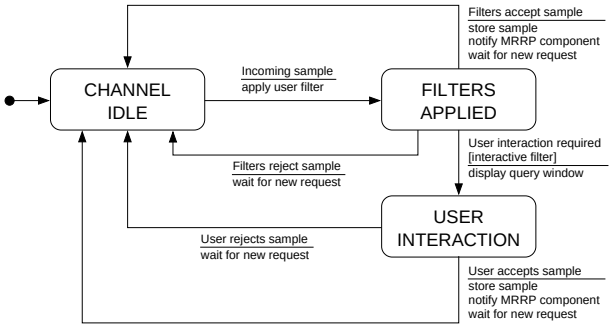

Figure 4.13: Minstrel Active Distribution Protocol (MADP) – Receiver side

As soon as a sample becomes available in a channel, the filters for that channel defined by the user (if any) are applied. If the sample passes the filters it is stored and the Minstrel Receiver Request Protocol (MRRP) component of the receiver is notified to take further action, i.e., request the corresponding shipment.
For some filters the user may want to make an automatic decision, when to accept a sample. For example, weather satellite images that are smaller than 50kB, come in three times a day and are free of charge are accepted while others violating any of these conditions are rejected.
For other samples the user may want to be asked. For example, a sample may announce special analysis data on stock markets at a price that is higher than the threshold configured for automatic acceptance. In this case the user's configuration may require user interaction to make the decision. Of course, this user interaction is not done synchronously as Figure 4.13 implies. Actually such jobs that require user interaction are queued and can be decided by the user at the time s/he wishes. But that would be an implementation view of the protocol engine, while Figure 4.13 gives the conceptual view.

4.7.2.1 Recipient-initiated Sample Distribution

For some receivers retries may be necessary frequently since they may not be reachable all the time. The most probable reasons for unavailability are:

- the receiving entity is not running (as a process at the receiving site)
- the receiver's site is not online all the time (dial-in users)
- network partitions

To improve support for such situations the concept of *receiver-initiated sample distribution* was included: A recipient can contact the MADP broadcaster via a special method call and request its queued samples. This triggers an extra user-specific queue run. The queue collects the samples queued for the requesting recipient and forwards them to the sender part which sends it to the requester.
This concept is similar to the mailbox functionality known from email where a user can retrieve his/her email from a mail server via the POP [116] protocol. It remedies several problems:

- It may occur that a recipient is online generally, but never exactly at the times when a sample distribution is attempted.
- Support for dial-in users is improved. If they are online infrequently, they can decide when to receive samples. However, this scheme no longer guarantees timely notification.
- The load on the MADP sender is lowered because it gets a request from a recipient and the triggered distribution attempt is likely to succeed because the recipient has explicitly announced its availability.

4.7.3 Minstrel Receiver Request Protocol

Figure 4.14 shows the receiver side of the Minstrel Receiver Request Protocol (MRRP).
The receiver side of MRRP has two main threads of execution: the requester that is responsible for requesting shipments and the queue which is in charge of queuing open shipment requests, i.e., when the retrieval was delayed by the user or because the supplier of the shipment (SA) could not be reached or could not supply the shipment when requested.
The requester starts off when it receives a notification from the MADP receiver that a new sample is available whose corresponding shipment is to be requested. In order to control retrievals the user may define a policy when and under what conditions a shipment is actually requested from the SA. For example, for some shipments it may suffice to be requested at some later time when the network is less loaded or rates for dial-in accounts are cheaper. If the application of the policy evaluates to "later retrieval" the request is queued. If the policy evaluates to "immediate retrieval" or no policy is defined, the shipment is requested immediately.
First the protocol engine tests whether the SA is reachable. If so, it requests the shipment from it. Upon successful receipt of the shipment, it is stored and the user is notified of the availability of new data. However, the retrieval of the shipment from the SA may fail for various reasons:

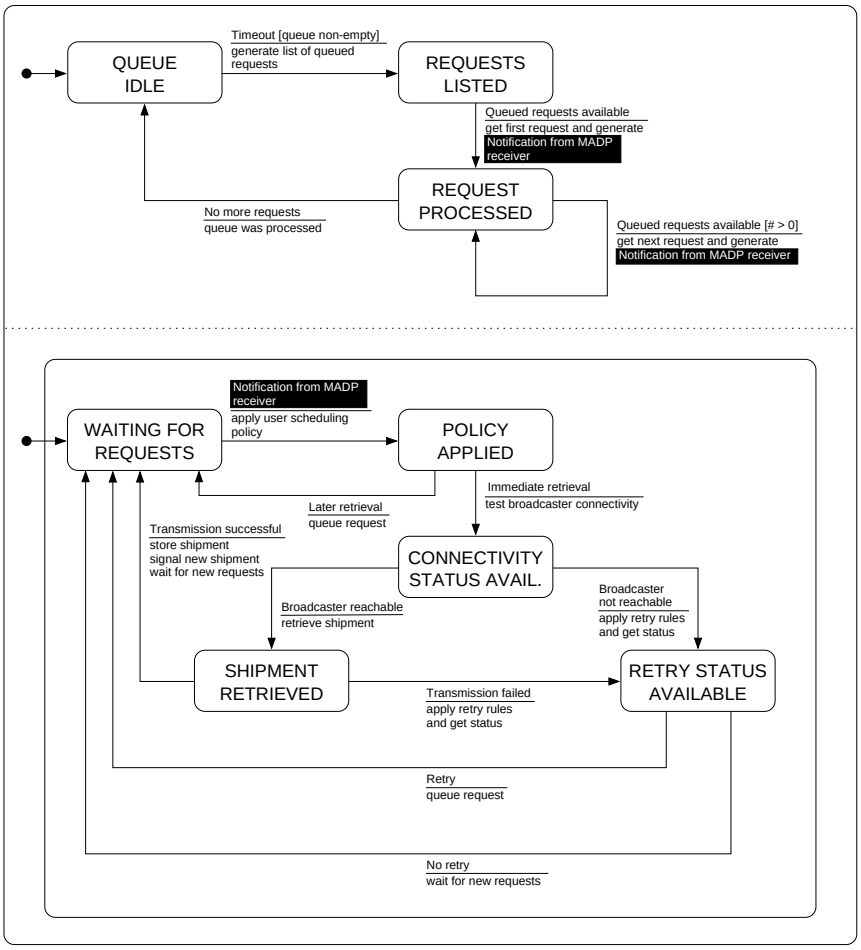

Figure 4.14: Minstrel Receiver Request Protocol (MRRP) – Receiver side

- low-level failures such as network disconnects, or
- application level policies and failures.

The second case indicates that the retrieval was refused or interrupted intentionally. For example, the SA may be a cache that does not already hold the requested shipment. In the case of a large shipment over a slow connection, the SA may decide to instruct the requester to abandon the retrieval and try again at a later time (the SA can supply an estimated delay to the requester) in order not to block the requester too long. In the meantime the SA could retrieve the shipment, so that it is available when the client issues a new request for that shipment. Other administrative decisions may be communicated in a similar way.
In any case of failure the retry rules are applied. These rules take into account variables, such as shipment expiry, and configuration settings, such as the maximum number of retries. Depending on the result, the retrieval is either queued or discarded.

The MRRP queue operates in parallel to the requester protocol automaton. Its operation is based on regular intervals: A timeout occurs after a configurable amount of time and the queue generates a list of retrieval requests. For every entry in this list, a notification is generated that triggers the retrieval process in the same way as if a notification from the MADP receiver were received after reception of a sample whose corresponding shipment is to be retrieved.
The generation of the list of retrieval requests is not fixed and can be extended in several ways. For example, every retrieval request can have a special delay attached that defines the delay for the retries instead of using a fixed interval. Such enhancements, however, do not change the queuing semantics, but only provide runtime optimizations.
Figure 4.15 shows the UML state chart for the broadcaster side of MRRP.

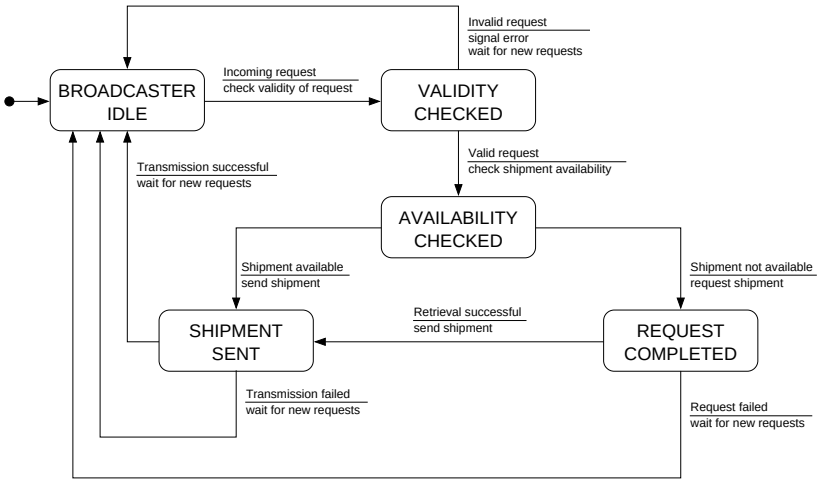

Figure 4.15: Minstrel Receiver Request Protocol (MRRP) – Broadcaster side

Every incoming shipment retrieval request is first checked for its validity, e.g., whether the requested shipment has expired. If this check fails, an error is signaled to the requester. Otherwise the availability of the shipment at the processing site is checked. In the case of a Minstrel Broadcaster, this check will always succeed. For a BDC, however, it may be necessary to request the shipment from its SA first. For example, if the BDC is a cache, a shipment will be available only after the first successful retrieval request.
If the shipment is not available at the processing site, it is requested from the site's SA. This may be signaled to the original requester as described above, for example, to instruct it to request the shipment at a later time when it is available. If the retrieval fails at all, the problem and its description are signaled to the original requester. Otherwise, if the shipment has been retrieved successfully or is already available, it is sent to the requester.
The interaction scheme of Figure 4.15 resembles very much the operation of a standard web server or of a web proxy server in combination with a web server.

4.7.4 Discussion of the Protocols

The main goal in the design of the Minstrel protocols was to support efficient operation while still keeping the protocols reasonably simple. Sophisticated protocols that maintain a high degree of data coherence and transmission security usually have very complicated protocol automatons that require a high volume of administrative data to be exchanged. They also tend to suffer severely from elaborate error recovery procedures in the case of failures. Thus the design rationale for MADP and MRRP follows the tradition of many successful Internet protocols and carefully trades powerful capabilities for conceptual simplicity. More features could have been added to the protocols (and actually were tested), but the gain would have been to low compared to the complexity introduced. The capabilities of MADP and MRRP seem reasonably complete for their domain.

As noted above, MADP and MRRP are based on RMI [66]. The ease of use and the excellent integration of RMI into Java come at the cost of lower efficiency. Compared to plain socket connections, less control over the timing and volume of data transmissions is available for RMI. For MADP this means that the typical sample size of 3.5kB as used in the examples splits into approximately 1kB of raw data and approximately 2.5kB of RMI overhead (administration data, serialization information, etc.). This overhead is mainly due to the sophisticated structure of samples that requires much class information to be transmitted with every call while the actual data tends to be rather small. The overhead/data ratio is not very good because samples try to be as concise as possible and thus are in a range, where administrative data has a higher impact on the total transmitted data volume. With larger sample sizes—which are undesirable—the overhead/data ratio would be far better. For MADP, however, the overhead seems to be tolerable due to the advantages of RMI.

An early implementation of the protocols tried to cut down on the reachability checks. In an early version of MADP, for example, receivers registered with their SA. The intention was that the SA would know what receivers were online and thus be able to decide quickly to whom to send a sample (without further connectivity checks) while queuing the sample for currently offline receivers. This assumption, however, was proven wrong since the concept of registration introduced the problem that a state was shared between several parties and needed to be kept consistent. For example, a receiver could crash or intentionally be stopped without de-registering at its SA. The same problem arose if the receiver was unreachable due to network problems. Maintenance of state information in the presence of such error conditions that frequently occur in the networking area had a far lower gain than what would have been won. Thus this concept was abandoned.

4.7.5 Possible Improvements

This sections briefly discusses some possible improvements for future implementations of the protocols.

A problem of RMI is that data objects sent via RMI must be built up in memory before they can be sent, received, or used. This is not a problem for samples that are rather small, but may be one for large shipments. If a shipment of, say 3MB, is requested by 10 users in parallel from one

SA, this may lower the performance of the SA. A solution may be to distribute shipments in a different way: The actual shipment class would hold only the administrative data and a pointer (a URL) where the cargo of the shipment could be retrieved. If a simple socket-based protocol were used for the cargo transfers, it would also allow recovery of interrupted transmissions. An interrupted transmission could restart at the point of interruption without everything's being re-transferred, as necessary with RMI.

For further optimization, connections could be reused for the transmission of multiple objects. This strategy, however, has to be applied carefully to prevent starvation effects. For MRRP the application of connection reuse is simple and comes at low cost. Shipment requests can be sorted according to SA and the connections can be reused. For MADP it can have dramatic impact and require complex scheduling strategies. If an MADP broadcaster sends all samples for one receiver in bulk over a reused connection, it may easily lead to starvation of other receivers if the sample rate is high enough. To prevent starvation effects, specialized scheduling strategies must be applied. At the moment, however, it is not clear whether this would pay off.

Chapter 5

The  MINSTREL Push System: Components

Minstrel follows the component and communication model presented in Chapter 2. This chapter describes the main components of the Minstrel system in detail. Section 5.1 describes the receiver, Section 5.2 presents the broadcaster, and finally, Section 5.3 gives a description of the base distribution component (BDC).

5.1 Receiver

The Minstrel receiver allows the user to access the Minstrel system. It offers a graphical user interface that allows the user to access channels (store, retrieve, display, and search shipments, samples, and offers) and manage his/her channel subscriptions. Via the receiver the user can manipulate, control, and customize his/her user profile, the received information, and the channels as well as update the configuration. Based on a channel's defaults and the user's settings the receiver is responsible for updating channel content, deleting expired data, and freeing disk space if necessary. This section gives an overview of the receiver and describes its main components in detail. Figure 5.1 shows the architecture of the Minstrel receiver.
The main components of the receiver are the Presentation Unit, which is in charge of displaying shipments, the Data Store Unit (DSU), which stores all channel content and associated administrative information, and the Minstrel Receiver Control Unit (MRCU) that manages the receiver's operation, controls the Presentation Unit and the DSU and interacts with the Minstrel infrastructure via MADP and MRRP.
Intentionally, Figure 5.1 does not show the e-commerce components, the Minstrel Data Lock (MDL) authentication infrastructure, and the graphical user interface.
The e-commerce and MDL components are conceptually outside the the Minstrel core components (receiver, broadcaster, BDC) as can be seen from Figure 4.1. The Minstrel receiver is a client of the e-commerce infrastructure to effect payment transactions and therefore is only aware of the e-commerce infrastructure's interfaces and some of its components which can be accessed from the outside. For example, the receiver knows how to issue a *pay* request to the e-commerce

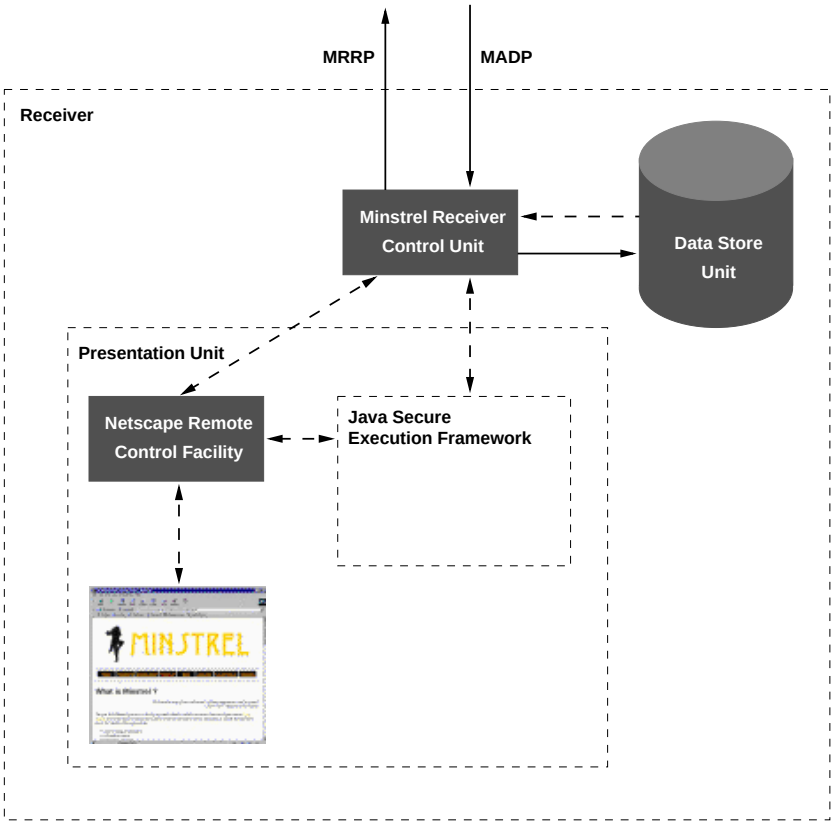

Figure 5.1: Architecture of the Minstrel Receiver

infrastructure or can retrieve the user's account information from it (the *wallet* which holds the user's "money"). This separation of concerns allows the receiver to work without the payment infrastructure if payment is not required. The infrastructure has a flexible payment interface that allows multiple business models and payment methods to co-exist. As a proof of concept, the current implementation of Minstrel supports a pay-per-view business model using the Millicent micro-payment protocol [54] as payment method. Minstrel's generic payment model and its payment infrastructure are described in Section 6.2. In-depth descriptions of micro-payments in Minstrel based on Millicent are given in [71] and [138].

Similar arguments apply for the relation between the receiver and MDL. Actually, MDL is intended to be invisible for the receiver and the other Minstrel components. Each sample or shipment is guaranteed to be authenticated and checked for tampering before being made available to a receiver or BDC. MDL and the Authentication and Verification Environment (AVE), which provides an high-level interface to MDL's functionalities, are integral parts of the Minstrel distribution infrastructure. The receivers, broadcasters, and BDCs are application-level users of this infrastructure and do not have to concern themselves with the low-level security details. Minstrel's security architecture and MDL are described in Section 6.1. A detailed presentation of MDL is is given in [43] and [44].

The graphical user interface of the receiver on the other hand is not shown in Figure 5.1 because it focuses on user interaction rather than Minstrel functionality. Though it interacts with all of the above components directly or indirectly, it "only" provides a means for the user to access functionalities of Minstrel components. Thus it is not described further.

5.1.1 Minstrel Receiver Control Unit

The Minstrel Receiver Control Unit (MRCU) manages the receiver's operation and interacts with the Minstrel transport system. It receives samples and can request the corresponding shipments from an SA. This means that the MRCU includes the implementations of the MADP and MRRP protocols described in Section 4.7.
Samples and shipments along with administrative data are stored in the receiver's DSU. Upon request of the user the MRCU can retrieve the data stored in the DSU and instruct other receiver components to operate upon it, e.g., instruct the presentation unit to display it.
Conceptually the MRCU is the main "contact partner" and dispatcher of information, both for the user and for the Minstrel infrastructure. It mediates between the user's requests and the Minstrel system's functionalities. If the receiver were an operating system the MRCU would analogously be its scheduler. The user interacts with the MRCU via the graphical user interface.

5.1.2 Presentation Unit

The presentation unit is in charge of presenting channel content to the user, e.g., by displaying text or graphics, playing audio content, or executing pushlets and agents, as described in Section 4.5.
The main requirement for the presentation unit is that it must be able to handle a wide range of different content (MIME) types. An abundance of such MIME types already exists on the Internet and new ones are emerging rapidly. This makes it nearly impossible to offer support for all of them. However, Minstrel was intended not to constrain users in their choice of content formats. Thus the compromise for Minstrel's receiver was to choose an existing component-off-the-shelf (COTS), integrate it into the receiver, and let it handle the displaying of MIME types.
A search for such components revealed that the most powerful and versatile components for this purpose were web browsers themselves. Compared to other components, like SUN's HotJava component, browsers like Netscape Communicator [125] (or Mozilla [115]) or Microsoft Internet Explorer [111] have four major advantages:

- they are free of charge,
- they quickly incorporate functionality for handling new MIME types,
- many plug-ins for highly specialized MIME types, such as Apple's QuickTime [3] video format or RealAudio's [141] streaming audio format, are available, and
- most Internet users have them and are familiar with them.

The disadvantage, however, is that the integration of browsers into applications is difficult and usually cannot be very tight. Nevertheless the browser strategy was chosen for its undoubted benefits.
The presentation unit uses an off-the-shelf version of Netscape Communicator to display standard MIME content. In the integration process two directions of information and control flow between the receiver and Netscape Communicator must be considered:

To Netscape Communicator: For example, if the user wants to view a shipment, the MRCU first has to retrieve the shipment from the DSU and then instruct Netscape Communicator to display it. For this type of control and information flow, the Netscape Remote Control Facility (NRCF) [68] was developed.

From Netscape Communicator: Less frequently, the user will interact with Netscape Communicator and may want to initiate an action by the Minstrel system. For example, Netscape Communicator displays a list of channels and the user wants to subscribe to one by clicking on a hyperlink. This must be communicated back to the MRCU by using the receiver as Netscape Communicator's proxy. With appropriate configuration settings Netscape Communicator can tunnel all user requests through the receiver which then can decide what action to take (trigger a Minstrel operation or forward the request).

Minstrel channels can also hold executable (mobile) code in two manifestations: agents and pushlets. An agent is a specialized program for processing the content of Cargo objects (see Sections 4.4 and 4.5). A pushlet is mobile code (and possibly some attached data) that is sent as a shipment. "Displaying" a shipment that holds a pushlet actually means that the pushlet is executed at the receiver. For both types of mobile code a special runtime environment must be provided that protects the receiver from erroneous or malicious code and ensures the user's privacy. The Java Secure Execution Framework (JSEF) of Minstrel offers such a runtime environment and allows the user to protect his/her local resources. The user and system administrator can define access rights and a security policy that is enforced by JSEF whenever an agent or a pushlet executes. The security issues concerning execution of pushlets and agents and Minstrel's Java Secure Execution Framework are presented in Section 6.1.3 and detailed descriptions of JSEF and its underlying concepts are provided in [70] and [84]. Thus the following description focuses on the integration of Netscape Communicator via NRCF.

5.1.2.1 Netscape Remote Control Facility

Several possibilities exist to control Netscape Communicator remotely:

- Platform-dependent
  - Write a new plug-in to control the browser (similar to the plug-ins available for PDF files, RealAudio, etc.); a specialized plug-in must be written for every supported platform.
  - UNIX: the `-remote` parameter

  - MS Windows: NCAPI (DDE and OLE)
  - MS Windows: `shelexec.exe` and `url.dll`
  - Apple: Macintosh remote control

- Platform-independent
  - LiveConnect: supports access from Java to JavaScript [127] (and vice versa) that provides a high degree of control over the browser
  - Java applet: the browser loads an applet that can in turn control it

After careful consideration the Java applet approach was chosen since it is platform-independent and does not impose further requirements. NRCF works as follows:

1. A special Java applet is loaded by Netscape Communicator.
2. This applet can instruct Netscape Communicator to load a certain URL via the standard method `showDocument()` of `java.applet.AppletContext` (this is the context every applet executes in).
3. To allow remote control, the applet registers with a remote controller that can send display requests (URLs) to the applet, which in turn instructs Netscape Communicator to display the URLs.

Figure 5.2 shows an example to demonstrate the interaction patterns.
First an `NRCFController` is created. This includes an `NRCFServerObject` that is in charge of managing subscriptions of clients (i.e., web browsers, remotely controllable via the `NRCFApplet`) and sending commands to the subscribers. Once an `NRCFController` server is running it can be contacted via RMI and clients (`NRCFApplet`) can register themselves (`subscribe()`). If the browser is to display something, a "SHOW URL" command is sent to the server that asynchronously sends the request to all subscribed applets, which in turn instruct their associated browser to display the data. Remote control of the browser can be enabled or disabled at any time (ENABLE and DISABLE commands in Figure 5.2). If the controlling server exits (OFF) all clients are notified asynchronously (`serverExit()`).
Servers (like `NRCFController`) that want to control Netscape Communicator remotely implement the `RemoteController` interface, while applets (like `NRCFApplet`) that want to register with a server and are in charge of receiving requests and controlling their associated Netscape Communicator implement the `RemoteControllable` interface. Both interfaces extend `java.rmi.Remote` and thus both `NRCFController` and `NRCFApplet` are RMI servers (see Figure 5.3).
When an `NRCFApplet` registers with `NRCFController`, it actually registers itself as an object of type `RemoteControllable`. This in turn requires that it implements this interface and specifies that it can be called via RMI. For example, the `showURL()` and `serverExit()` methods of `NRCFApplet` in Figure 5.2 are called via RMI.

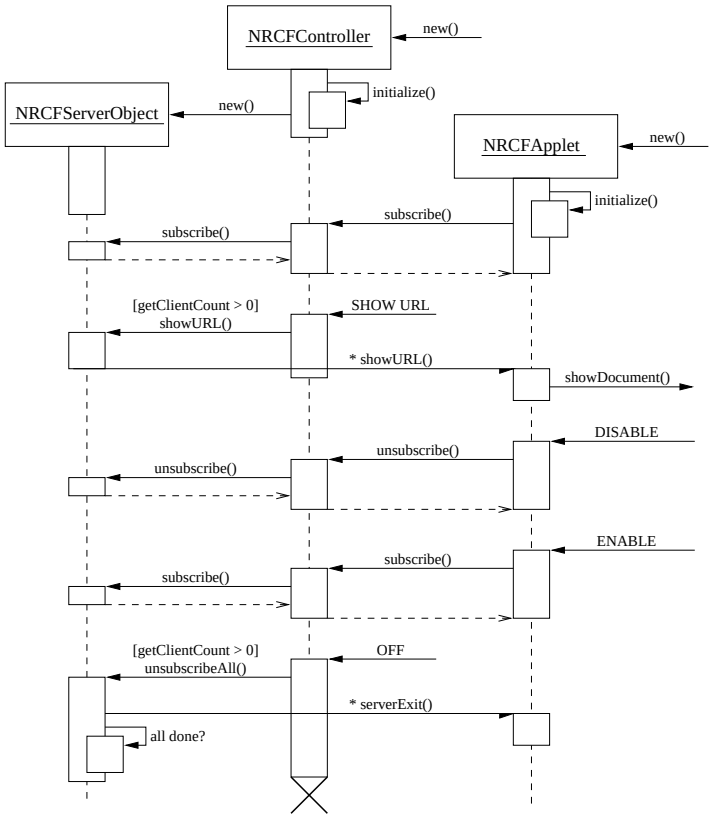

Figure 5.2: Operation of the NRCF (UML sequence diagram)

Actually, all communication between `NRCFController` and `NRCFApplet` is done via RMI. This was a main goal of the implementation of NRCF, in order to support a higher-level communication paradigm than plain sockets, which are used by most applets. However, this introduced some severe difficulties since Netscape Communicator does not include a bug-free and up-to-date Java Virtual Machine (JVM) that supports RMI as specified by SUN. Thus NRCF relies on the Java plug-in [159] supplied by SUN (free of charge) that can replace Netscape Communicator's JVM.
When Netscape Communicator encounters an `APPLET` HTML tag, such as,

```
<APPLET
 CODE="NRCFApplet.class"
 WIDTH="320" HEIGHT="100">
 <PARAM NAME="URL" VALUE="//:2020/RemoteControl">
</APPLET>
```

the applet uses Netscape Communicator's JVM, even if the Java plug-in is installed. However, if Netscape Communicator reads the following `EMBED` HTML tag,

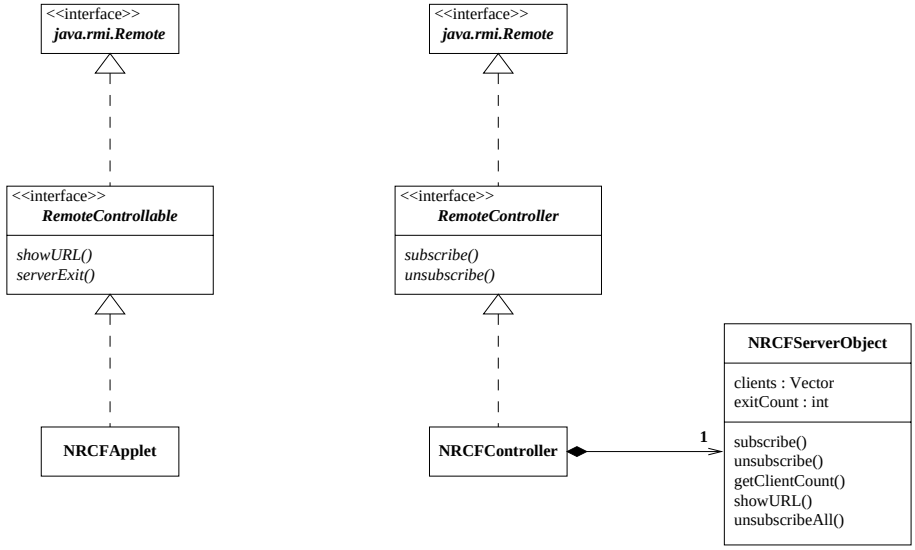

Figure 5.3: UML Class diagram of NRCF

```
<EMBED
 TYPE="application/x-java-applet;version=1.2"
 PLUGINSPAGE="http://java.sun.com/products/plugin/1.2/plugin-install.html"
 CODE="NRCFApplet.class"
 WIDTH="320" HEIGHT="100"
 URL="//:2020/RemoteControl">
</EMBED>
```

the applet uses the JVM from SUN instead. This allows applets to be written, that can use the newest version of SUN's JVM.
NRCF is also available as a separate package independent of Minstrel. It can efficiently be used in teaching systems, for example, where a teacher guides students through material by remotely controlling the students' browsers.
The implementation of NRCF can be adapted easily to work with other web browsers, too. Basically, any web browser that supports "remote control" via Java applets, such as Microsoft's Internet Explorer, which was also evaluated, could be used. Netscape Communicator was given the preference, however, and used in Minstrel, since it also is available for non-Microsoft platforms.
A further description of NRCF is given in [68].

5.1.2.2 NRCF Security

Remote control of an application, as NRCF can control Netscape Communicator, always introduces security problems. This is especially true in this case, since NRCF like all similar applets can be accessed and distributed over the network. This potentially allows anyone familiar with the applet's interface to control the browser of a user and indirectly access the user's machine with the user's access rights.

Such a security hole cannot be tolerated—though most implementations simply ignore it—and must be handled by a feasible authentication and authorization scheme. The first approach during the implementation of NRCF was that the applet started up an RMI server that implemented the RMI methods to be called to remote control the browser. By the use of certificates/passwords and session keys, only dedicated programs would gain access to the remote control methods. This was a rather heavy-weight concept for a very simple and frequently used functionality. Thus a better and much simpler solution was sought and is now implemented in NRCF.
Actually NRCF's approach does not require authentication and authorization at all, because of a very simple trick. Instead of letting the applet start a publicly available RMI server, it creates a private RMI server object and registers a reference to this object at the party that is allowed to control the browser. The remote control methods are only callable via this RMI server object. Thus, only the party which has explicitly been contacted by the applet can call its RMI methods, obviating the need for authentication and authorization.

5.1.3 Data Store Unit

The receiver's Data Store Unit (DSU) is the central database for storing channel content and administrative information (e.g., subscriptions). It implements efficient and flexible storage and retrieval strategies.
The important requirements for the DSU are:

Scalability. The DSU must be able to handle large amounts of data. For some channels, for example, image or movie channels, or software distribution channels, shipments can be huge.

Powerful searching. A simple yet scalable searching facility is required. Users want to find the information they have received in channels quickly and in an associative way regardless of the amount of data in the DSU.

Expiration. In order not to consume unlimited amounts of disk space the DSU must have a configurable and intelligent expiration facility that prevents the user's harddisk from filling up.

The DSU also must be adaptable to a wide range of application scenarios like periodicals, news tickers, or high-priority information channels. Typical scenarios for the DSU are:

Periodicals. Periodic magazines typically have several articles in each issue which can have references to other articles in earlier issues. References to future issues are also possible (announcements). Besides these, other relations exist: Depending on the publication interval a certain number of issues forms a volume, or there may be errata that correct errors in previous issues. While the publication interval may be rather long and thus easy to handle the important issue here is the relations among the objects which must be supported by the DSU.

News tickers. News ticker services are the other side of the spectrum: only tiny bits of information with a high publication rate whereas relations/dependencies between the information units are rare.

Replacing Updates. Some channels will hold content that outdates as soon as new content becomes available. New content will always override old ones and only one up-to-date copy of content exists at any given time.

The DSU must have a flexible design to support all these application scenarios. The following sections describe the design decisions taken to fulfill these requirements. The DSU is described in detail in [166].

5.1.3.1 Data Storage

Inside the DSU data is stored on a per-channel basis. Each stored channel consists of an index to support efficient access to the stored data, and a physical storage that holds the channel's administrative data and content (samples, shipments, etc.). Figure 5.4 depicts this design.

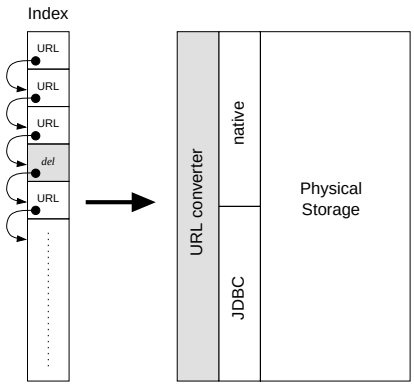

Figure 5.4: A stored channel

The index contains URLs that uniquely identify the data objects stored in the physical storage. They provide a uniform way to access the stored data, both for the human user and for other receiver components that need to access the DSU. URLs were also chosen to support bookmarking and provide a meaningful direct access method to human users. The use of URLs additionally would allow the user to access several physical storages—be it local or accessible via the network—and view those as one integrated entity.
For accessing the storage, the URLs are translated by the *URL converter* into calls to the physical data storage (native calls or via JDBC). Typical URLs have the following form:

```
minstrel://channel?attr=value?attr=value ...
```

In addition to native access methods to physical storage structures, JDBC [64] was taken into the design to provide a further layer of abstraction that supports the easy exchange of the physical data storage.

While this high-level design is straightforward, the question of what physical storage to use requires some discussion. The experimentation with database access via JDBC showed that access via the native interface of a physical storage component must also be supported, for the following reasons:

- At the moment JDBC is only offered by databases (although it is not restricted to databases). Thus if a JDBC interface were required, this would exclude other physical storages, such as plain file systems which are well-suited in many settings.
- Many databases have problems with Binary Large Objects (BLOBS). However, BLOBS, such as images or software, are a very common data type in push systems which must be supported well.

Tests with several JDBC drivers and public domain databases like MySQL and mSQL [184] showed that neither JDBC nor the databases worked well with BLOBS. Commercial databases like Oracle, Informix, or MS Access were not considered because they either are too costly (e.g., Oracle) or tie the storage to one architecture (e.g., MS Access). Thus a file-system-based solution was developed. JDBC, however, was kept in the design to be open to further developments in the database area.
Actually, the file system is well-suited for the DSU's purpose. The file system can store large binary data and provides fast and efficient retrieval mechanisms. Many other systems that require similar functionality as Minstrel rely on the plain file system, for example, the Squid web proxy [176] or Usenet News [90]. The main issue in all such systems is that information must be able to be found quickly and stored efficiently whereas database-like manipulations like joins or set operations should be very rare.
If the file system is used as physical storage, two requirements must be considered: (1) not too many objects, i.e., files, must be stored in a single directory and (2) a separate index must exist to search and retrieve the data objects efficiently. (1) is necessary because directory access is rather slow for large directories. However, it can easily be achieved by a hierarchic scheme as shown in Figure 5.5. (2) requires the implementation of an index structure that supports efficient searching. For Minstrel it is also necessary that this index supports searching for many different attributes and provides fast insertion and deletion of entries.
Figure 5.5 shows the structure used in the DSU for storing a large number of data objects in the file system.
This physical storage structure meets the above requirements and thus provides reasonably fast access to the data objects. It consists of a 3-level hierarchy of directories with the leaf directories holding the actual data objects, and has an efficient index that provides fast access and manipulation operations. With this 3-level architecture of the DSU's physical storage and a typical maximum of 1024 directory entries in each directory (e.g., in most UNIX file systems), 1024^3 leave directories can hold up to $1024^3 * 1024 = 1,099,511,627,776$ data objects. This is more than can possibly be used by a single user. However, this design is generic so that the number of directory levels can be increased to support even higher numbers of data objects.

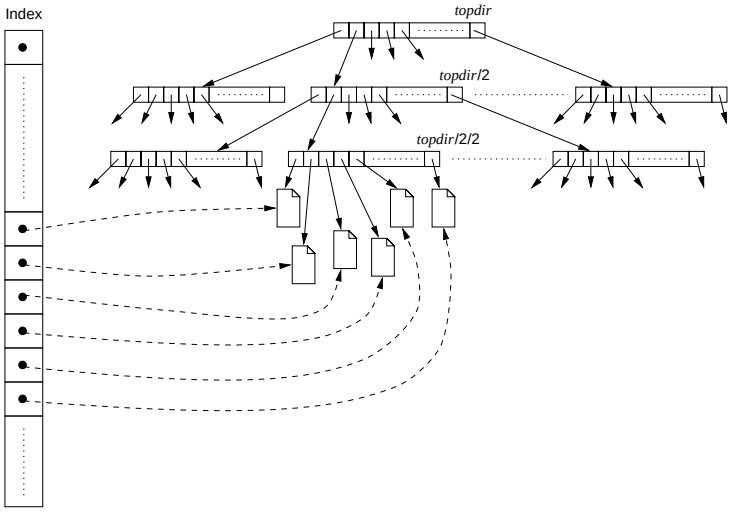

Figure 5.5: The file system as physical storage for the DSU

5.1.3.2 Indices and Searching

The index of the DSU actually consists of several specialized index objects. The two primary ones are a time-based index that lists data objects in the order in which they were received and an attribute-based index that allows quick search operations on many attribute values. Additional higher-level indices can be built on these two basic indices, for example, to support bookmarking of search results and relations between data objects. Figure 5.6 shows the basic design of the indices.

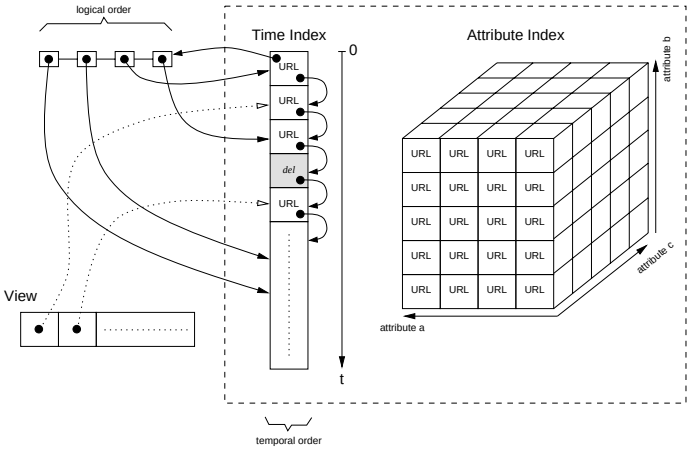

Figure 5.6: DSU indices

The time-based index supports queries like "list all data objects received between time *x* and time *y*." Besides this temporal order, data objects can be related in other ways. For example, issues

of a periodic electronic magazine can be related to form a volume. This is shown by the logical order in Figure 5.6. The attribute index (depicted only for 3 dimensions in Figure 5.6) provides access based on (combinations of) attribute values or attribute ranges. Views represent results of queries as references to one of the primary indices.
While the implementation of the time-based index and the higher-level indices is straightforward, the implementation of the attribute index is worth further discussion. The attribute index must support fast and efficient search access on many attribute values in parallel, its performance must be reasonable even in the presence of a constantly growing amount of data to be indexed, and it must not consume very high amounts of storage.
The requirements for the attribute index are:

- fast access time (constant if possible)
- adding of new objects in constant time
- reorganization (deletion, restructuring) must scale well
- symmetric multi-key access, i.e., there are no primary or secondary keys and fast searching for all key attributes is supported
- efficient interval search.

For a symmetric *n*-dimensional multi-key search space, an *n*-dimensional bitmap theoretically would be necessary, each "point" denoting a possible combination of attribute values. The size of the bitmap is the product of the cardinalities of the attribute domains. Since the attribute domains of samples and shipments are large, the bitmap would require a huge amount of space. Fortunately the bitmap is sparse and can be compressed to a reasonable size by the use of special data structures.
A search for existing data structures reveals that not many data structures exist that can fulfill the above requirements. Even commercial databases use rather "old-fashioned" data structures. The ones considered were K-D-B trees, splay trees, several other trees, and Grid files. Grid files [129] fulfill the above requirements and were chosen for the implementation of the attribute index. The assumptions for the application of the Grid file concept hold for the receiver's DSU: the indexed objects (shipments, samples) have a small number of attributes ($\sim$10), each attribute has a large domain, attribute values can be linearly ordered, and the attributes are independent of each other. Grid files compress the search space by partitioning it into *grid partitions* depending on the clustering of data points. Grid partitions are mapped onto *grid blocks* that are assigned to *buckets* which hold the actual data records. Figure 5.7 shows this for a 2-dimensional search space.
Inside one bucket records are stored in linear order. The DSU uses 8kB buckets. A record in a bucket holds administrative information about the indexed data and a reference into the physical storage where the data object is stored, i.e., a URL. The mapping between grid blocks and buckets is done via the so-called *grid directory*. Figure 5.8 shows a grid directory for a 2-dimensional record space.
The access strategies of Grid files operate upon these base concepts. Grid files are a highly complex but very efficient data structure. Their application in the implementation of the attribute

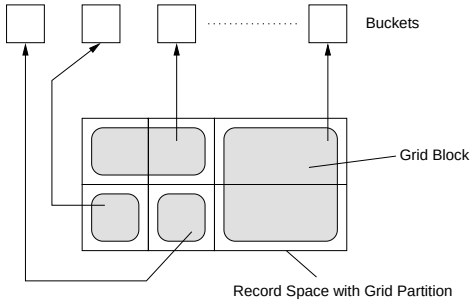

Figure 5.7: Partitioning of the search space and assignment to buckets

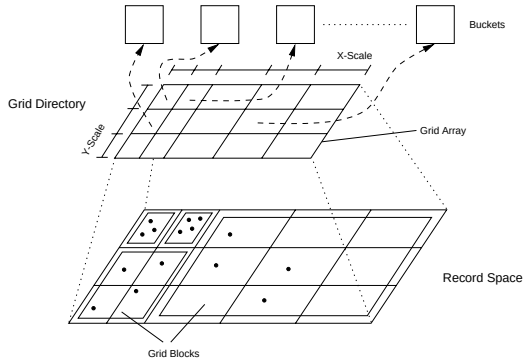

Figure 5.8: The grid directory

index of the DSU is described in [166] along with a detailed description of Grid files and their access algorithms.
By the use of an attribute index based on Grid files, efficient and powerful search capabilities are available. Part of the DSU's user-friendliness depends on these capabilities because searches will occur frequently. All interesting attributes of the Minstrel data objects are indexed. Search operations are done via filling out a form in a Query-By-Example (QBE) manner. This allows the user to specify search conditions in an intuitive yet powerful way. Other access types (insertion, deletion) are done in a similar way. Further details are provided in [166].

5.1.3.3 Content Expiration

Minstrel—like any other push system—may have transported huge amounts of data to every receiver after being in operation for some time. The user's harddisk will steadily fill up with channel content if no administrative procedures are applied. The strategy for deleting unwanted or expired data chosen by most push systems is to make the user responsible for deleting such information and ensuring that not too much disk space is consumed.
Minstrel's DSU takes a more user-friendly approach by supporting the definition of an expiration strategy enforced by the DSU. On basis of the validity information contained in samples, ship-

ments, orders, etc., and a channel's configuration, the DSU can expire content. Additionally, the user can define customized expiration rules that override these system rules. For example, the user may not want content to expire even if it is outdated, if it was expensive or is still of interest. The definition of expiration rules works in an intuitive way as shown in Figure 5.9.

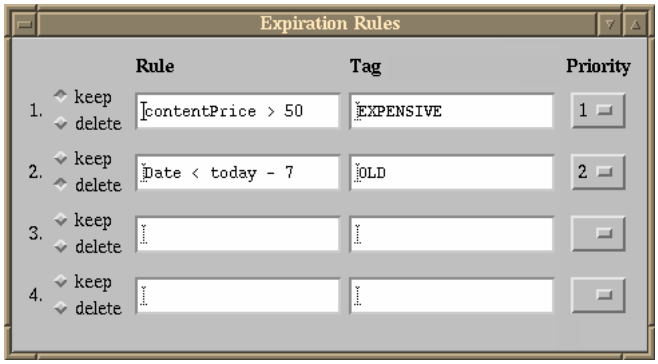

Figure 5.9: Declaration of expiration rules

Expiration rules can be tagged with a name and given a priority. In Figure 5.9, for example, the user defines a tag EXPENSIVE that marks all channel content that was expensive and defines that it should be kept. The second rule OLD defines what the user considers old content which could be deleted. When an expiry run is performed, all EXPENSIVE content will be kept even if it is OLD, whereas other OLD content will be expired.
The implementation of the expiration strategy is described in detail in [166].

5.2 Broadcaster

Minstrel employs a combination of the *primary broadcaster* and *simple broadcasting* approaches described in Section 2.2.2: The broadcaster is the primary source of channels and relies on the transport infrastructure consisting of BDCs for the distribution of channel content to large numbers of users.
Figure 5.10 depicts the architecture of the Minstrel broadcaster.
The architecture consists of the following main components:

- the Source Update Facility (SUF) which is in charge of interacting with information sources and feeding channel content received from the data sources into the channels,
- the Subscription Management Unit (SMU) which handles subscription requests and manages the subscription database,
- the Data Store Unit which stores all channel content and management information such as administrative data of channels, subscriptions, or receipts for payments, and

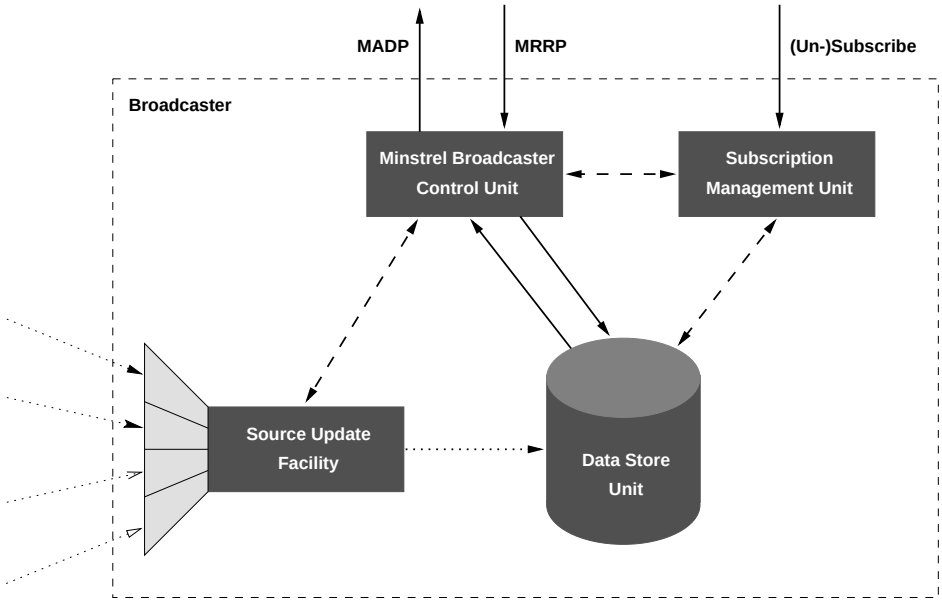

Figure 5.10: Architecture of the Minstrel Broadcaster

- the Minstrel Broadcaster Control Unit (MBCU) that manages the broadcasters operation, controls the other broadcaster components, and interacts with the Minstrel transport system via MRRP and MADP.

Figure 5.10 intentionally does not show the e-commerce components, the Minstrel Data Lock (MDL) authentication infrastructure, and the management interface (and its GUI). The reasons for this are the same as given for the Minstrel Receiver in Section 5.1.

5.2.1 Minstrel Broadcaster Control Unit

The Minstrel Broadcaster Control Unit (MBCU) manages the broadcaster's operation and interacts with the Minstrel transport system. Via MADP it sends samples to subscribers and satisfies incoming shipment requests via MRRP. The samples and shipments it handles are stored in the DSU along with related administrative data.
Besides distributing channel information, the MBCU also supplies the administrative interfaces that are necessary for managing the operation of the broadcaster. It collects statistical data that allows monitoring of the broadcaster's operation. Logging data is provided to support traceability of the broadcaster's operation and serve as the basis for problem analysis. On the basis of this information the administrator can tune the broadcaster's operation or take action if problems occur.
The MBCU also supports monitoring of MADP and MRRP and enables the administrator to manually influence the protocols' operation. For example, the administrator can view the queued samples or initiate a sample distribution attempt. Via the MBCU the administrator can also

change protocol parameters such as timeouts, retrial intervals, or the queuing policy. Additionally the MBCU provides an interface to startup and shutdown the broadcaster.

5.2.2 Data Store Unit

The broadcaster's DSU is the central database for storing samples, shipments, administrative data of channels, subscriptions, etc. Its architecture is similar to that of the receiver's DSU but simpler. While the receiver's DSU is tuned towards flexibility and user-friendliness (searching, expiration), the broadcaster's DSU is targeted at information maintenance and speed.
The basic design is identical to the receiver's DSU as described in Section 5.1.3. However, due to the different requirements, the broadcaster's DSU does not need as powerful indexing and searching mechanisms as the receiver's DSU, which simplifies its architecture and implementation. The broadcaster trades sophisticated searching capabilities for fast storage manipulation and retrieval.
The requests the broadcaster has to satisfy all bear unique identifiers that uniquely denote the information requested (samples, offers, and shipments), which facilitates direct access via one designated attribute (similar to a primary index in a database). Indices on other attributes are only necessary as far as administration is concerned. Thus only a few attributes, such as the validity fields of the data, require fast access and are indexed. Additionally, a time-based index is maintained which is simple to implement and comes at nearly no cost. These side conditions make the indexing for the broadcaster's DSU much less elaborate and require much simpler data structures than those necessary for the receiver's DSU.
Content expiration is an even more important issue for the broadcaster than it is for the receiver since it has to handle higher amounts of data. While the receiver may subscribe only some of the channels a broadcaster offers and thus only needs to deal with that data, the broadcaster stores information for all its channels. The expiration strategies for the broadcaster's DSU thus are more focused on the professional, administrative user who typically requires less sophisticated but more powerful interfaces which also support highly-configurable and fully automated operation.

5.2.3 Source Update Facility

The Source Update Facility (SUF) is the broadcaster's interface to data sources. Data sources provide content which they want to have distributed via one or more of the channels that the broadcaster distributes. In order to support flexible data injection, the SUF has a flexible, modular, and extensible architecture that can mediate between data sources and the broadcaster.
The SUF has a generic RMI interface that provides all functionality necessary to inject content into channels. This interface can be used directly or may serve as the basis for domain-specific interfaces that facilitate content exchange with data sources such as databases or web servers. For a database, for example, a small database-specific interface can be implemented and added to the SUF, which then can receive content and accompanying administrative data from the database and feed it into channels via the RMI methods of the generic SUF interface as shown in Figure 5.11.

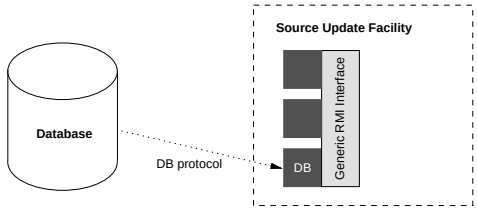

Figure 5.11: SUF interfaces

The generic SUF interface can be used from programs but also supports content injection via a graphical user interface. The GUI allows the user to specify the new content and define the required administrative information such as description, validity, or price. Part of such a dialog to insert new content into a channel is shown in Figure 5.12.

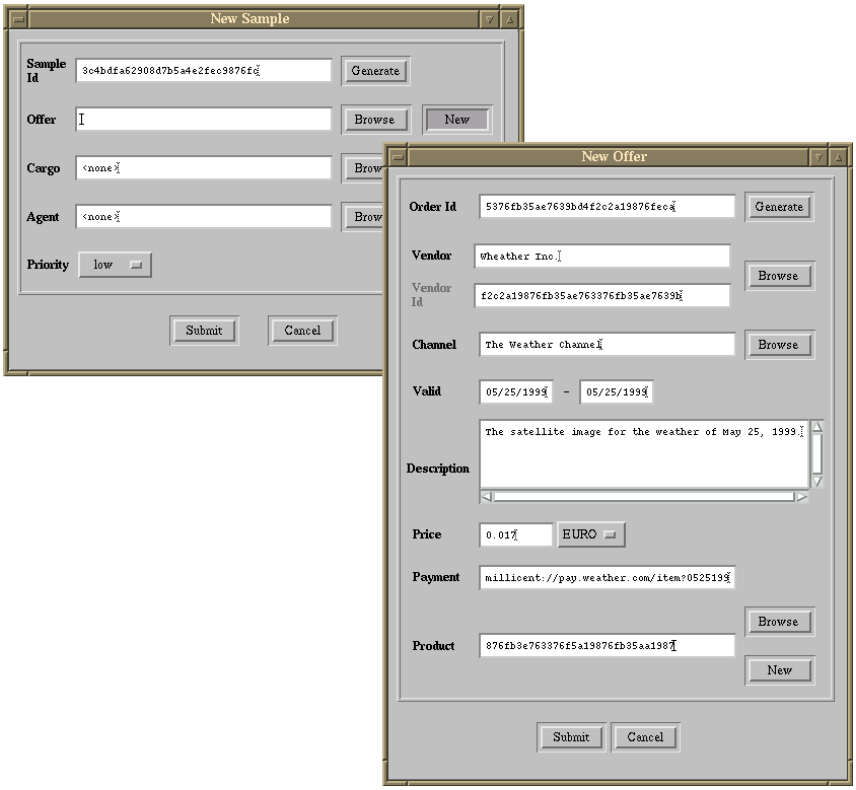

Figure 5.12: Definition of a new Sample

In this example, the user is defining a new offer as part of a sample which is to be sent over a channel. Later in this dialog the user will have to provide the content for the corresponding shipment. This is usually done by providing the URL where the SUF can retrieve the content.
Making the generic interface of the SUF available via RMI has several advantages. It can be used locally but also from remote data sources which is the standard case. At the site of the data source

a specialized graphical user interface may be available and transfer all necessary information to the SUF via RMI calls. Additionally, data sources can implement local components that mediate between the data source's requirements and the SUF interface as shown in Figure 5.13, which means that the data-source-specific interface runs at the data source's site instead of the broadcaster's site.

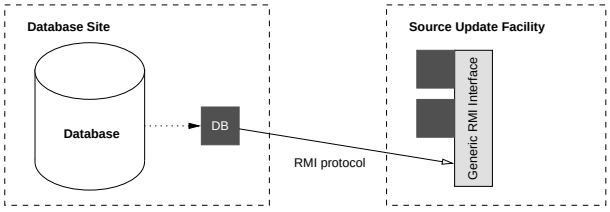

Figure 5.13: Local SUF interface

5.2.4 Subscription Management Unit

The Subscription Management Unit (SMU) handles (un-)subscription requests for channels and maintains the subscription database. Parties that want to subscribe to a channel interact with the SMU, where they can specify the channels they want to receive, provide their profiles if wanted, and define the level of privacy they require. At some later time this information can be modified, for example, changes to a party's profile can be made. The SMU likewise handles unsubscription requests.

The SMU maintains subscriptions in an area of the DSU where the MBCU can access it. The MBCU must be able to access this data to build up its distribution tables for MADP (see Section 4.7.2).

For reasons of privacy, subscriptions are maintained locally and not forwarded. Thus no party knows all subscribers and their profiles. Every subscriber—be it a human user via his/her receiver or a BDC—can specify a privacy policy that defines which information is private and which may be publicly accessed on request. If a party wanted this data, it would be possible to collect it, but only according to the policies defined by the subscribers. For example, if a broadcaster wanted to collect data about all receivers subscribed to one of its channels, it could issue a recursive request to its direct subscribers ($\Delta_{\mathcal{S}} = 1$), ask for their subscriber lists, and instruct them to reissue the same request to their direct subscribers. According to the policies defined by the subscribers the broadcaster may or may not get the data requested or only part of it.

5.3 Base Distribution Component

The Minstrel Base Distribution Components (BDCs) build up the Minstrel distribution infrastructure (see Chapter 4) which makes Minstrel's broadcasting strategy scalable to large numbers of subscribers. By using BDCs to build up a hierarchical broadcasting infrastructure, Minstrel

can guarantee timely deliverance (see Section 4.6.3) and lower the load and network bandwidth consumption of every node participating in the broadcasting process. Without BDCs, Minstrel would only be usable in small-scale settings.
A BDC can be viewed as a cross between a broadcaster and a receiver: It acts like a receiver towards its SA and as broadcaster towards its subscribers. Thus its architecture combines components of both worlds, as depicted in Figure 5.14.

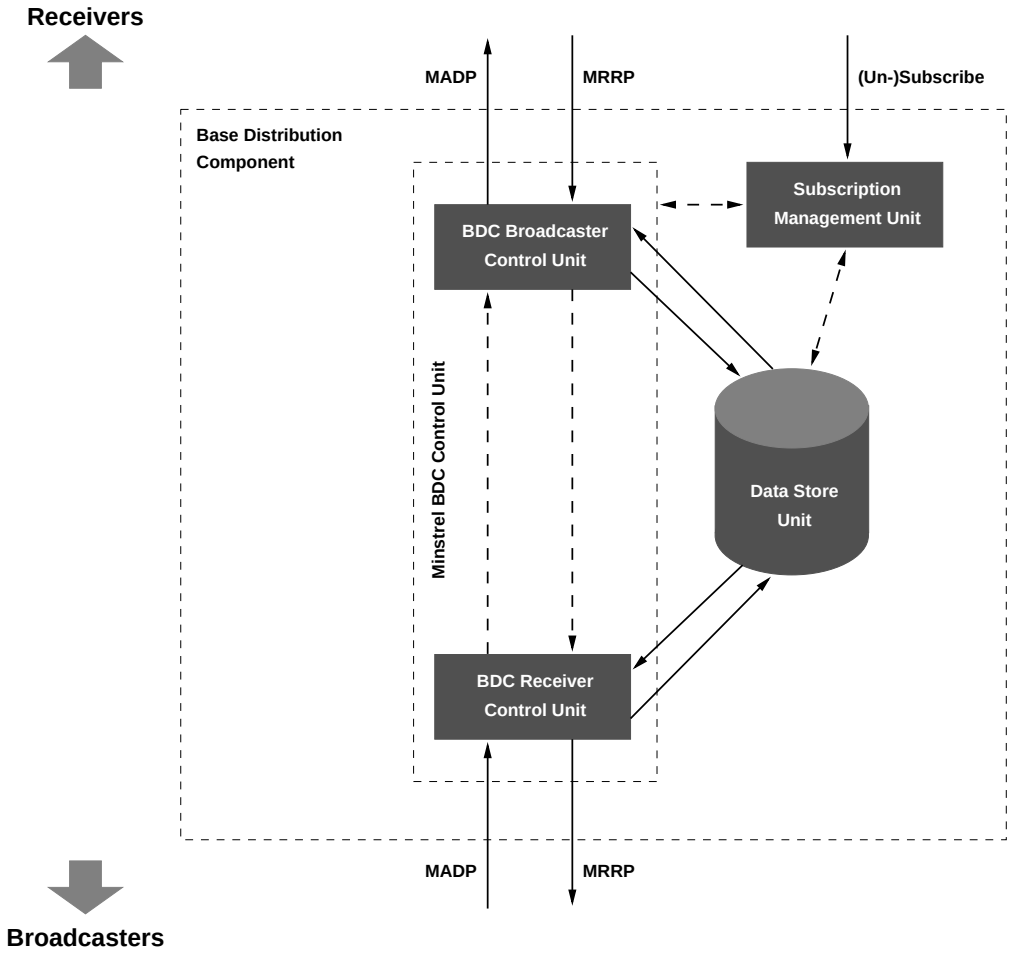

Figure 5.14: Architecture of the Minstrel BDC

From its broadcaster pedigree it inherits the Subscription Management Unit (SMU) to handle and maintain subscriptions and the DSU to store channel data and administrative information. The requirements for a BDC's DSU are identical to those of a broadcaster (see Section 5.2.2). Since it acts like a broadcaster towards its subscribers (other BDCs or receivers), its SMU also is identical to a broadcaster's SMU.
The control unit of a BDC, however, is hybrid. It consists of a receiver control unit and a broadcaster control unit that cooperate with one another and control the SMU and DSU. Basically the

MRCU and MBCU components can be reused but need to be extended to support collaboration between them. This, however, can be achieved easily.
Unlike the other components discussed so far, a BDC can be configured in three possible ways: as a repeater ("pre-loaded" cache), a cache, or a proxy. These configurations try to meet typical requirements in a network and mainly differ in their characteristics regarding the request time of a shipment via MRRP.
If the BDC is configured as a *repeater* then it immediately requests shipments whose corresponding samples it has received. This means that the receipt of a sample via MADP triggers two actions: (1) the sample is redistributed using MADP and (2) a request for the the corresponding shipment is issued via MRRP to the BDC's SA, after completion of which the shipment is stored locally. This interaction scheme implies that most shipment requests from one of the BDC's subscribers can be satisfied almost immediately (provided that the transmission of the relevant shipment to the BDC is completed before the shipment is requested). The repeater configuration targets environments where fast delivery is important. Thus shipments are cached at repeaters without waiting for a relevant client request.
However, automatic request of shipments may not always be necessary. In this case the BDC can be configured as a *cache*. Upon receipt via MADP, samples are redistributed but the corresponding shipments are not requested by the BDC. A shipment is only requested if an MRRP request is received from one of the BDC's subscribers. In this case the shipment is requested from the BDC's SA, stored locally and then sent to the requester. Further requests for the same shipment can be satisfied from the local copy. The cache configuration of a BDC targets environments where a certain delay is tolerable, while network bandwidth or disk space are limited, or user requests are not so predictable that a repeater would be justified.
A BDC configured as *proxy* facilitates access to channels and the Minstrel infrastructure where receivers cannot gain direct access, e.g., when receivers are located behind a firewall. Every proxy has a domain translator component that translates back and forth between the functionalities Minstrel requires and the application domain functionality. For example, in the case of a firewall it translates between the firewall requirements and MADP and MRRP, respectively.
By the use of BDCs the Minstrel distribution infrastructure facilitates caching implicitly as described in Section 4.6.5. This is in contrast to the world-wide web where a caching infrastructure had to be introduced in a second run, while it comes at nearly no additional cost in the case of Minstrel.

Chapter 6

The  Push System: Security and E-Commerce

Minstrel is intended to offer a platform for information commerce over the Internet. The two main requirements for Minstrel to be usable in such a business environment are authenticity and integrity of information, and support for payment methods and business models. Recipients of information want to be able to rely on the received information and possibly use it as a basis for business decisions. This is especially important in typical push application domains like news agencies, financial information services, and other businesses for whom reliability of data is paramount. Thus proof of origin (data authenticity) and proof that the information has not been tampered with (data integrity) must be provided. These proofs also provide the foundation for limitation of legal liability.

Section 6.1 presents Minstrel's authentication infrastructure which facilitates authentication of information origin and integrity checks through digital signatures. Moreover, Section 6.1 also addresses mobile code security, since Minstrel supports the distribution of pushlets and agents, which are executed at a receiving site and can threaten the security and integrity of a system. Thus Minstrel includes a highly configurable secure execution framework for Java code which protects a system from erroneous or malicious code and ensures its integrity, security, and privacy.

Chapter 1 has already mentioned a number of application domains for push systems, such as news agency information systems or electronic newspapers. In several of these domains the information provider will want to make revenue out of its business, and will charge for the information it disseminates over a push system. Thus push systems must support payment methods and business models to be applicable in such environments. Surprisingly, the issues of payment and e-commerce are not addressed by any existing push system so far.

To remedy this obvious shortcoming, payment facilities have been included in Minstrel. Minstrel offers a flexible and generic payment model that can be used to implement a variety of business models, such as pay-per-view or volume-based. The payment model decouples the business model employed from the underlying payment method(s), so that (theoretically) arbitrary payment methods can be used. Section 6.2 describes Minstrel's support for e-commerce. Its generic payment model is presented and an instantiation of this model using a micro-payment scheme is described.

6.1 Security

Security is relevant in several respects for push systems:

Authenticity and integrity of information
Information that comes over push channels should be authenticated. The receivers of the information must be sure that the information they get comes from the source it claims to and that it was not changed on its way from the sender to the receiver. This is especially important if information is to be charged for or must be especially reliable (for example, stock quotes or financial analyses).

Confidentiality of information
Channels can transport confidential information which must not be disclosed, for example, descriptions or analyses only for receivers who are known to the originator of the information. A similar problem exists with copyrighted content that is charged for.

Mobile code security
If the push system supports the delivery of code that is to be executed at the receiver's site, strict precautions are required to ensure the integrity of the receiver's site. The code must come from an authentic source (e.g., for reasons of liability), be prevented from spying out the receiver's private information, and be restricted from unlimited access to the receiver's resources.

The following sections describe how Minstrel addresses these issues.

6.1.1 Authenticity of Information

Minstrel Data Lock (MDL) is the basis of Minstrel's authentication infrastructure. It ensures that the information distributed over push channels comes from authenticated sources and has not been changed on its way from the sender to the receivers. Broadcasters and BDCs on the other hand can be sure that the requests they get come from authenticated clients, i.e., their channels are accessed by authorized receivers. MDL provides the following features:

- authenticated message origin (sender)
- integrity of messages.

These features are realized through digital signatures, a security mechanism based on public key encryption and certificates [148]. This protects Minstrel users against the two most problematic security attacks that endanger push systems:

Message tampering: interception and alteration of messages before they reach their intended recipients

Masquerading: sending/receiving messages using somebody else's identity.

The following sections give an overview of the architecture and concepts of Minstrel's authentication infrastructure. Detailed descriptions are provided in [43] and [44].

6.1.1.1 Architecture of the Authentication Infrastructure

Minstrel's authentication infrastructure has a layered architecture as shown in Figure 6.1.

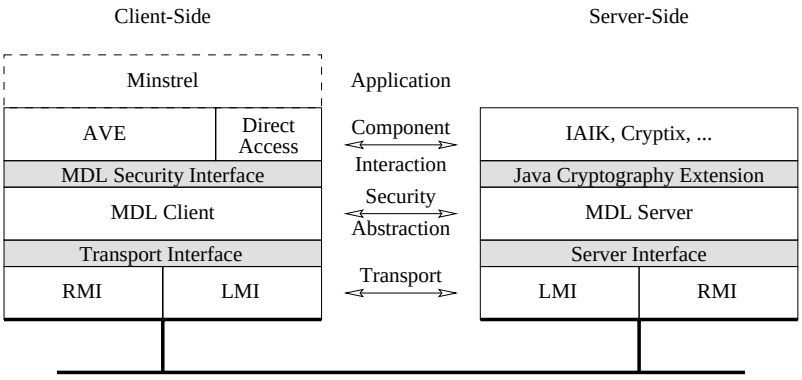

Figure 6.1: MDL layered architecture

From an architectural point of view the Minstrel components (broadcasters, BDCs, and receivers) are clients of the authentication framework (which of course is an integral part of the Minstrel system). Applications (clients) that want to use the infrastructure will primarily access it via the Authentication and Verification Environment (AVE) layer. AVE provides a specialized, easy-to-use, high-level interface to the infrastructure that does not require knowledge of the underlying concepts.

The functionality of AVE is composed around the concept of a *Guarantee*: by use of a Guarantee, objects can be digitally signed and verified (authenticated). A Guarantee can take any Java object, sign it and return the signed object as a *GuaranteedObject*, i.e., as an object whose origin and integrity can be proven. A GuaranteedObject holds all information necessary for later verification of the object; the receiver of the GuaranteedObject can easily do this by feeding it into its local Guarantee. The main design goal of AVE was to provide an interface that makes authentication as simple as generating a dedicated object (Guarantee) and calling some of its methods so that the whole process of authentication can be done in a few lines of Java code (from the application's point of view).

If the application wants to have more detailed control of the signing and verification processes, it can also directly access the Minstrel Data Lock (MDL) layer, which provides the necessary security abstractions (MDL must be trusted by all parties). Clients access this layer through the Minstrel Data Lock Client (MDL Client). MDL provides a client-server infrastructure for authentication: All signing and verification requests of higher layers are managed and directed to a MDL Server that provides the requested functionalities. Conceptually this means that the MDL Server is a specialized application server that implements the actual signing and verification processes.

This client-server architecture makes MDL very flexible and allows the user to choose a configuration that best fits the local requirements. The two most common configurations will be that (1) users have an MDL Client that uses a local MDL Server or (2) a set of MDL Clients access an

MDL Server that runs on a dedicated host (e.g., one per LAN). These configurations are hidden from the MDL layer through the Transport Interface and the Server Interface, respectively. In case (1), requests are satisfied via library-like Local Method Invocation (LMI) calls; in case (2), Remote Method Invocation (RMI) is employed.
The signing and verification functionalities of the MDL Server are implemented on the basis of the Java Cryptography Extension [85] and rely on the X.509 certificate standard [79]. For fulfilling its task the MDL Server also needs a Certification Authority (CA). A CA is a trusted third party that guarantees the identity of certificate bearers (like a passport guarantees the identity of a person). CAs, however, are outside Minstrel.

6.1.1.2 The Minstrel Authentication Process

This section gives an overview of the authentication process in Minstrel, the parties involved, and their interaction patterns. Detailed descriptions of this complicated process—both technically and organizationally—are provided in [43] and [44].
As described above, all communication in Minstrel is authenticated. For example, samples and shipments are digitally signed and their integrity, authenticity, and freshness (via the timestamps of the authenticated objects) can be checked by any involved party. Figure 6.2 gives an overview of the parties in this process and their interactions when a piece of information is being *verified*.

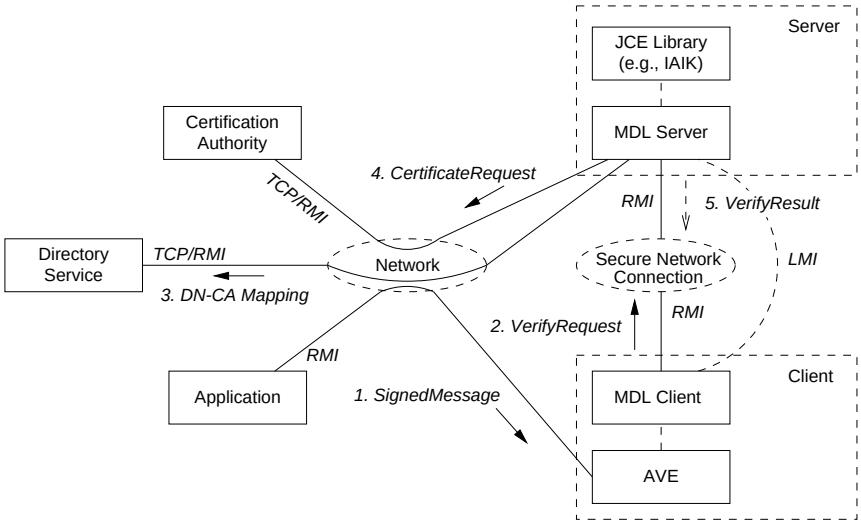

Figure 6.2: MDL components and their interactions during verification

Verification means that an application, e.g., a Minstrel component, has received a piece of information—a *message*, or in terms of the implementation, a GuaranteedObject—and now wants to check whether it is authentic, has not been changed and is fresh. First the message is handed over to AVE, which generates feasible requests to its underlying MDL Client layer to request verification. If the configuration uses a local MDL Server (see Section 6.1.1.1) a local library call is issued to the MDL Server library. If a remote MDL Server is employed, these calls

are made to the server site via RMI. This communication must use a secure network connection, e.g., by employing one of the transport mechanisms mentioned in Section 6.1.2, to return a meaningful and reliable result.
The MDL Server must then retrieve the certificate of the message's (GuaranteedObject's) signer, which holds the signer's public key required for the verification procedure. Thus the server first has to contact a Directory Service to find the Certification Authority (CA) that holds and guarantees the signer's certificate. This CA is then contacted and the signer's certificate is requested.
Now the MDL Server can verify the message and return the result of this verification process to the client, which hands it over to the requesting party. Figure 6.3 depicts this process in terms of the implementation, showing the collaboration of the various objects.

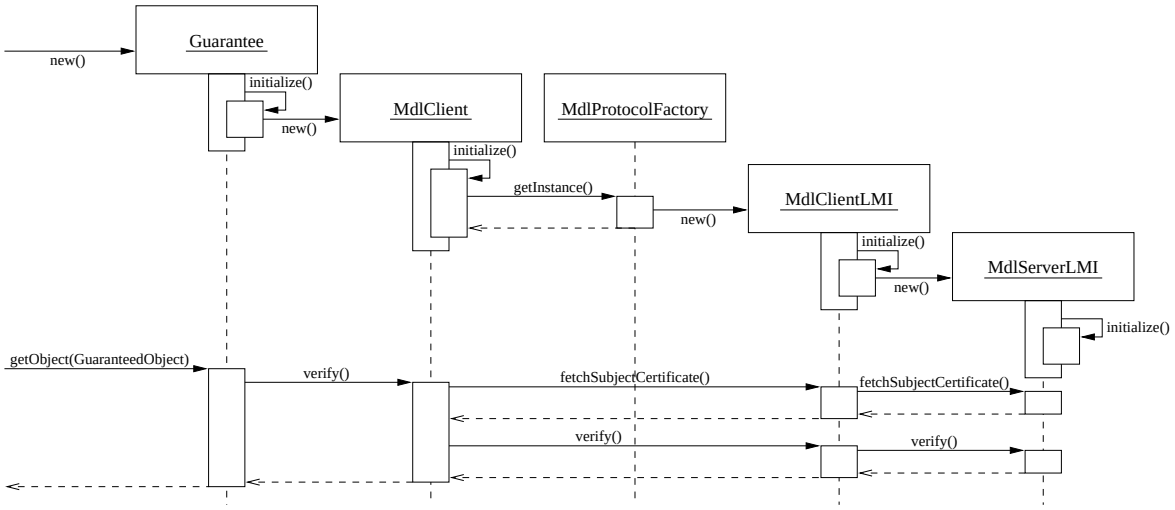

Figure 6.3: Verifying of information (UML sequence diagram)

The signing process works in a similar way, as shown by the sequence diagram of Figure 6.4.
The only major difference to the verification process is that the party requesting the signing process needs a public and private key pair and a certificate from a certification authority.
In the model of Figure 6.2 the directory service and the CA were viewed as two distinguishable parties. However, if feasible systems are available, these two roles can be unified. One such system that could satisfy both requirements is X.500 [17], which is likely to be employed for this purpose after the widespread acceptance of its Lightweight Directory Access Protocol [173] (LDAP). To be prepared for such developments, MDL uses a flexible, easily adaptable interface for accessing directory services and CAs based on the Java Naming and Directory Interface [160] (JNDI).

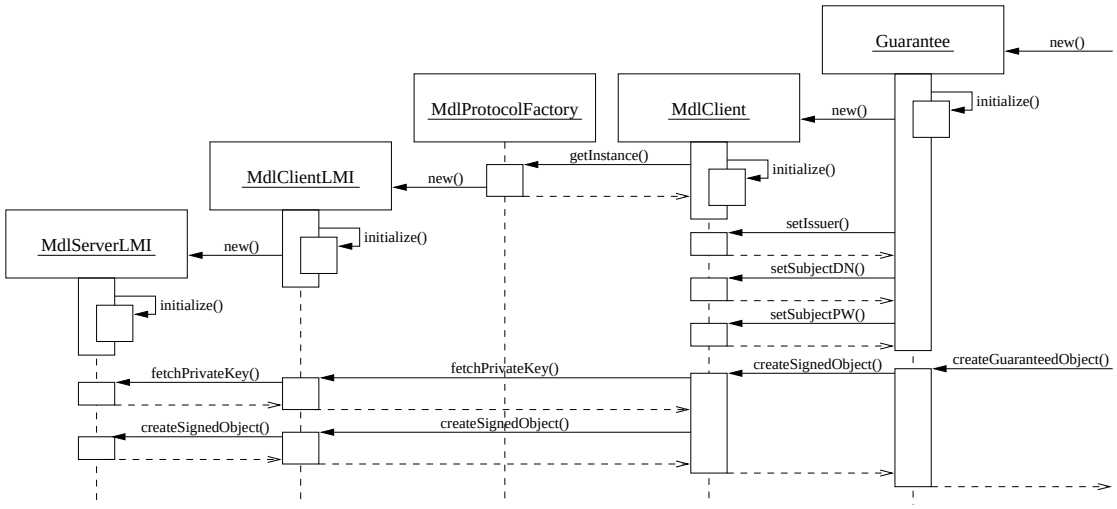

Figure 6.4: Signing of information (UML sequence diagram)

6.1.2 Confidentiality of Information

Confidentiality of information can be highly important depending on the application domain of a push system. For example, channels could hold confidential data or, in the case of a pay channel, the provider would want to prevent access from users who were not charged.
The standard technical concept for ensuring confidentiality is the application of encryption methods. For a push system this means that all channel content and administrative data is encrypted before being distributed. Minstrel, however, does not include means to ensure confidentiality, i.e., an encryption package, by default. This decision was taken, after much discussion and careful consideration of the advantages and disadvantages, for the following reasons:

Redistribution

Redistribution of information that was legally acquired, e.g., by a user who paid for it, cannot be prevented efficiently. Such a user has necessarily been supplied with means (keys, etc.) to decrypt the information and can easily redistribute the decrypted information. This is the same problem that challenges copyright owners in general (compare this to video cassettes, software, etc.). It can only be solved by digitally watermarking all the information that is distributed and by suing redistributors. Checking all information that is distributed places extreme demands on the provider in terms of time and money which far outweigh the benefit of copyright protection.

Tapping

The current Internet protocols (HTTP, TCP/IP, etc.) are not secure. Any data transmitted over such protocols can be tapped and run through a brute-force code-breaking attack.

Man in the middle

To be scalable, any push system's transport system includes concepts like caches or re-

peaters. Frequently such components are run by third parties, such as Internet service providers, which in turn then have full access to all the data.

Dedicated lines

Traffic between Internet service providers, on the other hand, is mostly run over dedicated, rather secure networks that are hard to attack (except by a provider's staff). A similar argument holds for receivers: Dedicated lines run by telecommunication companies are not easy to attack. For other access media like satellite or cable TV, which are broadcast media that can be received by non-authorized persons rather easily, protection can be effected with special encryption hardware. Special hardware for such access types is needed in any case, so encryption hardware could be included from the beginning.

Broken chain

Information that is transmitted encrypted must also be encrypted when stored at the receiver's site. Currently this is only supported by a small fraction of sites and depends on a site's configuration and operating system. In a world of PCs running simple Windows-style operating systems, this cannot be guaranteed at all. Such systems typically are a very easy target for intruders.

Pre-paid information

If receivers have to pay for channel information, it frequently means that they pay in advance (for example, a flat fee). Once they have paid for the information, i.e., they have acquired the right to get/use it, why should it be encrypted?

Mass media

Push systems are mass media. The financial damage from unauthorized access will be low compared to the effort put into investigation and protection. Compare this to the policy of many European countries, where consumers are required to pay a license fee to the public TV broadcasters if they own TV sets. Most people pay the fee, some do not, but the system works.

Despite these arguments, encryption cannot be ignored. For some application domains it will be necessary to support encryption techniques. Nevertheless it is likely that the decision of not including a dedicated encryption package into Minstrel will not have a major impact. Many components-off-the-shelf (COTS) to provide secure, encrypted communication can be used with Minstrel. Such components typically provide a secure communication substrate that can be used transparently with existing applications.

For example, Netscape's Secure Sockets Layer protocol [128] (SSL) which is the de-facto standard for secure Internet communication or its successor, the Transport Layer Security protocol [33] (TLS), can be integrated into Minstrel at nearly no cost. Since SSL supports any higher-level protocol that uses standard socket-based communication, a simple strategy would be to run all Minstrel protocols over SSL connections if secure communication is required (see Figure 6.5).

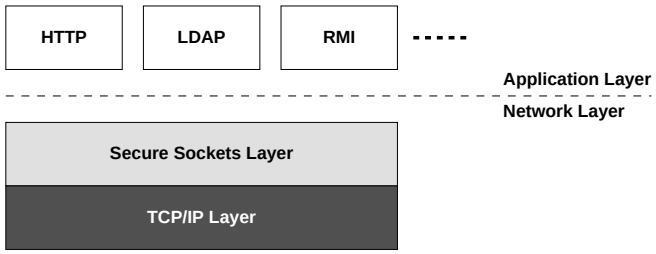

Figure 6.5: SSL runs above TCP/IP and below higher-level application protocols [128]

This is possible although MADP and MRRP are RMI protocols because RMI is built upon sockets and allows control of its socket layer via special factory classes.[1] Java implementations of SSL which can be used for Minstrel are available, for example, iSaSiLk [77], JCP SSL-Pro [80], or SSLeay [76].
Another advantage of using industry standard COTS over a proprietary Minstrel encryption implementation is the benefit of being able to use up-to-date implementations which can cope with most recent security threats, without the additional cost of code maintenance.

6.1.3 Mobile Code Security

Minstrel supports two types of mobile code [171, 178] that are closely related: pushlets and agents. An agent is specialized Java code for handling the content of Cargo objects (see Sections 4.4 and 4.5). A pushlet is mobile code (and possibly some attached data) that comes as content of a channel, i.e., as a shipment. Displaying a shipment that holds a pushlet actually means that the pushlet is executed at the receiver.
Though agents and pushlets add a high degree of flexibility to Minstrel and are an important part of modern distributed applications, they also introduce far-reaching security problems for the site that executes them. In both cases code that comes from an outside source is executed on the receiver's site, has the receiver's access permissions and possibly can harm the site's integrity, security, and privacy.
Agents and pushlets introduce four basic categories of potential security threats [23, 58, 109, 110]:

Leakage: unauthorized attempts to obtain information belonging to or intended for someone else

Tampering: unauthorized changing (including deleting) of information

Resource stealing: unauthorized use of resources or facilities (e.g., memory, disk space)

Antagonism: interactions not resulting in a gain for the intruder but annoying for the attacked party.

[1] `java.rmi.server.RMIClientSocketFactory` and `java.rmi.server.RMIServerSocketFactory`.

Thus a special runtime environment must be provided that protects the receiver from erroneous or malicious code—be it intentional or unintentional—and ensures the receiver's integrity, security, and privacy. Such a runtime environment is offered by Minstrel's Java Secure Execution Framework [70, 84] (JSEF), which is based on Java's security model. It allows the user to protect his/her local resources. The user and the system administrator can define access rights and a security policy that is enforced by JSEF whenever an agent or a pushlet executes.
The following sections discuss Java security issues, compare JSEF with the standard Java security model and give an overview of JSEF's architecture, concepts, and functionalities. In-depth descriptions of JSEF and its underlying concepts are provided in [70] and [84].

6.1.3.1 Java Security

Several approaches have been devised to cope with the security implications of mobile code. According to [146] four practical techniques for securing mobile code exist:

Sandbox model: This model restricts the privileges of the code to a limited set of operations.

Code signing: Code that comes from trusted sources is trusted and granted full access.

Firewalling: When code enters a trusted domain it is examined; the decision whether and how to run it is based on specific code properties.

Proof-carrying code: The mobile code carries a proof that it satisfies certain properties; this is a promising new technique that can currently be applied only in very limited settings.

Since Minstrel is implemented in Java and all agents and pushlets are restricted to Java code, the Java security model [50, 55, 56, 162] is the basis for all code security issues in Minstrel. Minstrel's security features can only work inside this general model.
Java uses a hybrid approach that combines sandboxes and code signatures. It is important to note that Java code signatures and the signatures used for information authentication in Minstrel (see Section 6.1.1) are similar but strictly separated concepts. Although in Minstrel all information (including agents and pushlets) is authenticated, i.e., has not been tampered with and comes from an authentic source, Java requires a specialized code signature and its verification as defined below. This may be overhead but cannot be eliminated due to the restrictions imposed by the Java security model. Every time the Java Virtual Machine [97] (JVM) runs a piece of code—a class file in Java's terminology—the following steps occur [55]:

1. The Java virtual machine obtains a class file and accepts it if the file passes preliminary bytecode verification [185].

2. The Java virtual machine determines the class's code source. This step includes signature verification, if the code appears to be signed.

3. The Java virtual machine consults the security policy and composes the set of permissions to grant to this class. In this step, the policy object will be constructed, if it has not been already.

4. The Java virtual machine loads and defines the class, and marks the class as having been granted the set of permissions.

5. The Java virtual machine instantiates the class into objects and executes their methods. Runtime type-safety check continues.

6. If at least one method of a class is in the call chain when a security check is invoked, the access control code examines the class's set of granted permissions. It does this to see if there is sufficient permission for the requested access. If yes, the execution continues. If no, a security exception occurs. When a security exception—which is a runtime exception—occurs and is not caught, the Java virtual machine aborts.

7. When the class file and the instantiated objects are no longer in use, they are garbage-collected.

On the basis of these concepts and processes, Java and thus Minstrel can prevent many security attacks but not all. As already mentioned the Java security model can only be applied to Java code. Security threats that are outside the Java environment cannot be controlled or even noticed by Java's security mechanisms. Only local (to the executing site) resources can be protected by these mechanisms, and security misconfigurations by the user that open security holes cannot be prevented.
Aside from these problems Java does quite a good job. It provides good protection against the two most dangerous threats of leakage and tampering, while the less dangerous ones of resource stealing and antagonism cannot be fully prevented. This, however, is due to the fact that it is hard to distinguish automatically between legitimate and malicious actions.

6.1.3.2 The Java Secure Execution Framework vs. the Standard Java Security Model

Though Java's security model provides strong mechanisms to protect the user from security threats introduced by agents and pushlets, it falls short when it comes to higher-level security configuration and its management.
Java's current security model only supports explicit specification of accesses that are permitted. This enables the user to specify all that is necessary to secure his/her site. It is not very practical, however, if the user wants to permit many different types of accesses. Instead of specifying what is permitted, frequently the opposite semantic is required, i.e., to specify what is *not* permitted. JSEF supports both ways of specification by its so-called additive and subtractive permissions.
The current security model of Java uses a two-level configuration approach. A global policy file holds the default permissions for any user on a specific site and a user's local policy file can specify additional permissions. Since Java's security model only supports additive policies, only two extremes for the security configuration exist: Either each user must have and maintain a private security policy file, or a global policy is specified and user-specific configurations are ignored. Either way has its shortcomings. With the first strategy users can easily introduce security holes—regardless whether a global policy file exists or not since the user's local policy can extend the global policy in any way—but can have a personalized configuration. In the

second case the administrator has total control over the security policy but cannot tailor it to specific users' needs.

JSEF overcomes these problems by providing a hierarchic security policy scheme that supports both local, user-specific security policies and a global security policy defined by the administrator that takes precedence over user policies. At runtime of a Java program, a user's actual policy is defined by merging the user's local policy with the global policy. The user's policy, however, cannot circumvent restrictions imposed by the administrator in the global policy. This scheme improves the management of security policies in a secure yet flexible way.

Moreover, the Java security model lacks support for user groups. JSEF on the other hand supports the definition of hierarchic user groups with assigned security policies. A user can be member of a set of groups that have different security profiles. With user groups being supported, an administrator can easily define a set of profiles in terms of groups and assign these profiles to users depending on the users' requirements. Additionally these groups can be freely structured into a hierarchy, which further supports maintenance and tailoring of the security policy.

JSEF supports the retrieval of policy definitions from arbitrary sources: JSEF currently uses XML files [14] but can easily be tailored to load policy definitions from other sources such as databases or remote locations. This can be accomplished by the use of specialized handler classes provided by the user of JSEF. Mobile code that is to be executed can also be loaded from arbitrary sources. A prototype blueprint for loading class files and JAR files from any location in the file system (not only from within the defined Java classpath!) is included in JSEF. This concept supports storing of mobile code in arbitrary locations and formats, for example, in a database.

In the standard Java security model the requester of an operation receives a security exception whenever an access is denied by the user's security policy. This typically terminates the execution. If this was not intended, the user has to exit the program that wanted to perform the access, set the appropriate permissions, restart the program and retry its execution. This can be tedious and time-consuming.

JSEF therefore provides a powerful and flexible security negotiation facility. If a forbidden operation is attempted, JSEF intercepts it *before* the actual access and starts a negotiation process. Currently this means that the user is asked via a GUI whether the access should be denied, permitted once, for the current session, or always and thus be entered into the user's security policy. This supports runtime management of the security policy while still ensuring that the existing policy settings are not violated. The interactive negotiation scheme can be used as a blueprint for other (semi-) automatic negotiation schemes.

JSEF is based on the Java security framework and extends it with the mechanisms listed above. The following sections provide a more detailed look on the main concepts and components of JSEF. A detailed description of JSEF, the definition of JSEF security policies, and implementation details are given in [70] and [84].

6.1.3.3 The JSEF Policy Concept

JSEF's policy concept is a subclass of the Java policy concept. Applications that rely on JSEF ignore the settings defined in the Java policy of a user (in the system policy and user policy files) and rely solely on the policy defined for JSEF. As already mentioned above, policy handler classes are used to abstract from physical storage and support the loading of policy definitions from arbitrary locations using arbitrary protocols.
JSEF introduces the notions of *additive* and *subtractive permissions*. Additive permissions are the class of permissions as used by the Java security model: They grant permission to access a resource. Subtractive permissions on the other hand are an extension to this, allowing specification of what resources are not permitted to be accessed. The collection of additive permissions makes the user's *additive security policy* and the collection of subtractive permissions defines his/her *subtractive security policy*. The subtractive policy always overrules the additive policy. For Java code to be allowed to access a resource, the following conditions must hold: (1) the access to the resource is not forbidden by the subtractive policy and (2) the access to the resource is explicitly allowed by the additive policy. If an action is neither forbidden nor explicitly allowed, it is prohibited.
The policy concept of JSEF distinguishes between a *global* (defined by the administrator) and a *local policy* (defined by the user) which both can hold additive and subtractive permissions. Global settings always overrule local ones. Thus a user cannot break the security policy defined by the administrator. The format for defining permissions is given in [84]. Figure 6.6 shows a sample local policy definition that grants all permissions to all code from *www.sun.com* signed by *CK*, but forbids write and execute access to the user's home directory hierarchy for such code.

```
<?xml version="1.0"?>
<!DOCTYPE localPolicy SYSTEM "localPolicy.dtd">

<localPolicy userName = "Charly Brown" lastChanged="03/21/1999">
  <addItems>
    <policyItem signedBy="CK" codeBase="http://www.sun.com/-">
      <permission class="java.io.AllPermission">
      </permission>
    </policyItem>
  </addItems>
  <subItems>
    <policyItem signedBy="CK" codeBase="http://www.sun.com/-">
      <permission class="java.io.FilePermission">
        <permissionName name="/home/-"/>
        <actions name="write execute"/>
      </permission>
    </policyItem>
  </subItems>
</localPolicy>
```

Figure 6.6: A sample local policy definition in JSEF [70]

As already mentioned, JSEF supports the definition of hierarchical groups. Each group can be assigned a set of permissions. Users then can be declared to be members of a set of groups

and thus have the permissions defined by the groups. Groups, their hierarchy, and their access rights are typically defined by the system administrator. Figure 6.7 shows some sample group definitions by the administrator for a user *Charly Brown* who inherits the permissions defined by group *User* and has the restrictions defined in group *Developer*.

```
<?xml version="1.0"?>
<!DOCTYPE usergroups SYSTEM "usergroups.dtd">

<usergroups lastChanged="03/21/1999" changedBy="CK">
  <user username = "Charly Brown">
    <addgroups>
      <group groupname = "User" />
    </addgroups>
    <subgroups>
      <group groupname = "Developer" />
    </subgroups>
  </user>
</usergroups>
```

Figure 6.7: Mapping of a user to groups in JSEF (group policy) [70]

JSEF strictly separates additive and subtractive groups and their hierarchies to support clear and intuitive configurations. This, however, does not constrain the power and applicability of this concept. Additive subgroups always extend the permissions defined in their parent group while subtractive subgroups further restrict them. Each group can have exactly one parent group of the respective (additive or subtractive) type. Figure 6.8 shows some sample group definitions that define a group *Admin* and a subgroup *Developer* of *Admin*.

```
<?xml version="1.0"?>
<!DOCTYPE groupHierarchy SYSTEM "groupHierarchy.dtd">

<groupHierarchy lastChanged="03/21/1999" changedBy="Clemens">
  <group groupName="Admin">
    <policyItem codeBase="http://www.sun.com/-">
      <permission class="java.io.FilePermission">
        <permissionName name="/tmp/-"/>
        <actions name="read"/>
      </permission>
    </policyItem>
  </group>
  <group groupName="Developer" parentGroup="Admin">
    <policyItem signedBy="CK" codeBase="http://www.sun.com/-">
      <permission class="java.io.FilePermission">
        <permissionName name="/projects/-"/>
        <actions name="read write execute"/>
      </permission>
    </policyItem>
  </group>
</groupHierarchy>
```

Figure 6.8: Some sample group definitions in JSEF [70]

Admin defines that unsigned code from *www.sun.com* may only read the */tmp* directory hierarchy. Group *Developer* inherits this permission and extends it by additionally granting full access to the */projects* directory hierarchy to all code from *www.sun.com* if it was signed by *CK*.
The definition of the group hierarchy and the assignment of permissions to groups is part of the global policy specification which also defines the mappings between users and groups. At runtime JSEF merges the local and global definitions to determine the current access permissions, which are then enforced by the JSEF security manager. The processing of an access request is shown in Figure 6.9.

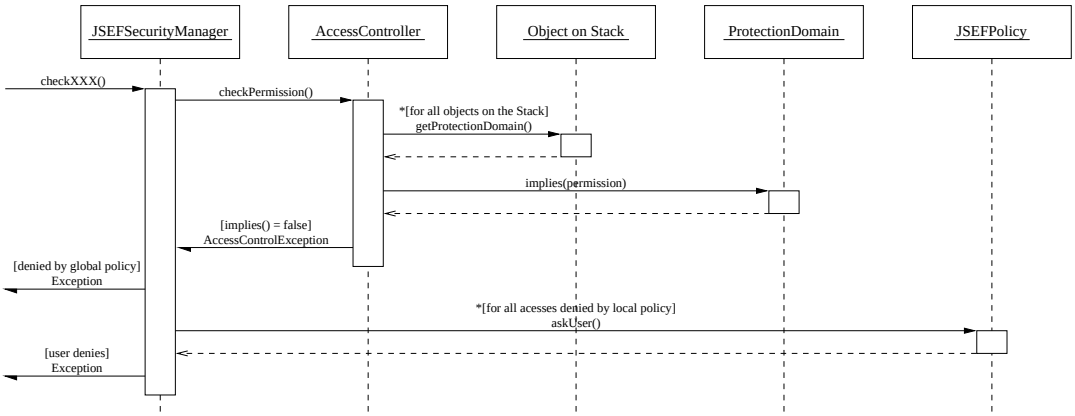

Figure 6.9: Processing of an access request in JSEF (UML sequence diagram)

In this process global definitions overrule local ones and restrictions are always stronger than explicit permissions. JSEF's security manager asks Java's `AccessController` to check a certain permission. This object in turn iteratively requests the protection domain of every object on the stack (stack inspection). The protection domain of a Java object is a collection of permissions associated with this object and holds a specialized `JSEFPermissionCollection` object that knows how to deal with the JSEF policy concept (additive and subtractive permissions, local and global policy, groups). Then the access controller tests whether the access is allowed, and if this check is positive the stack inspection continues. If not, an exception is raised. JSEF's security manager checks whether the access denial was due to a rule in the global policy. If yes, the access cannot be allowed and an exception is raised. If a local policy rule was the reason for the denial, the security manager records this and instructs the access controller to continue with the stack inspection. This continues until the access controller has processed all objects on the stack or the violation of a global policy rule aborts it. Now the security manager has a list of permissions that were not yet granted and were not forbidden by a global policy rule. If the list is empty the access is granted. Otherwise the user will be asked to decide whether to grant or deny the required permissions. If the user denies, an exception is raised; otherwise the access can be performed.
In-depth descriptions of the JSEF policy concepts are given in [70] and [84]. The processing of access requests is presented in detail in [84].

6.1.3.4 The JSEF Process

Figure 6.10 shows the main steps of the process every piece of mobile code has to run through until it may be executed. At runtime the accesses of the executing code are monitored by JSEF.

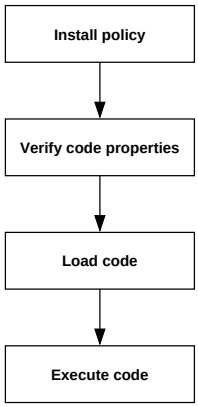

Figure 6.10: The JSEF Process

At the moment mobile code in Minstrel is either a pushlet or an agent and is restricted to Java classes contained in Java Archive (JAR) files. After such a piece of code, which is to be executed, has been downloaded, a secure environment in which this code will execute must be set up as a first step. This environment consists of a user-defined policy, a global policy defined by the administrator that overrules a user's settings, and a security manager that cooperates with an access controller to enforce the security policy.
This means that the actual permissions the mobile code will have at runtime must be determined in this step. The process merges the security policy defined by the user with the general security guidelines set up by the administrator as described in Section 6.1.3.3. The result of this process is a *policy object* that represents the combined additive and subtractive policy settings of the user-defined and the system-wide policies. The merging process is described in detail in [70] and [84].
This policy now has to be enforced by the security manager. Since JSEF includes concepts that extend the standard Java security policy, a specialized JSEF security manager is used. The JSEF security manager is implemented as a subclass of the standard security manager. It is instantiated and installed for the mobile code execution environment with the above security policy settings. At runtime the JSEF security manager will check every access of the executing mobile code (see Figure 6.9). It is interesting to note that in JDK 1.2 the security manager is included mainly for compatibility reasons with earlier versions. The actual checks are performed by the access controller (see Figure 6.9).
At this point a secure execution environment has been created and the second step of the process can start. In this step the properties of the mobile code must be checked and verified. This introduces some difficulties since in Minstrel all information is loaded from the receiver's SA, which is not the source of the information. Since the source of mobile code determines what

access permissions a piece of code has, the origin of the code must be explicitly stated in the administrative information of the code (the JAR file's manifest). Additionally, the administrative information must include the signer of the JAR file and the issuer of the signer's certificate. With all this information the origin of the code can be proven (in cooperation with MDL) and the signature of the JAR file can be verified (this is required by Java's class loading mechanism [55]). The manifest also explicitly names the "main" class that is to be used to start the code in the JAR file.
Their properties having been proven, the classes in the JAR file can be loaded in step three. The loading is performed by a special JSEF class loader that extends the capabilities of the standard Java class loader [55] with functionalities needed by Minstrel such as loading of classes from arbitrary locations.
Finally, in step four, the mobile code can be started by calling the standardized start method of its "main" class. During its execution, the mobile code will be monitored by the security manager and access controller that were set up for the execution environment in step one. Every access to resources will be intercepted and checked as shown in Figure 6.9.
In some cases, when a forbidden access is detected, the user will be asked whether to allow the access. The user can deny it, allow it once, for the current session, or always. In the last case the new permission is added to the existing ones and stored in the user's configuration. If the user denies the access or the access was denied by the global policy settings, the executing code will receive an exception indicating a security violation.
Further details on the concepts used in JSEF and the problems encountered during its implementation are given in [70] and [84].

6.2 Electronic Commerce and Payment

One of Minstrel's goals is to create a substrate for information commerce over the Internet. Indeed, *payment methods* and *business models* have to be addressed by any commercial Internet system. Because of the existence of the subscription phase in Minstrel, standard solutions such as macro-payments or flat fee systems (e.g., monthly charge to credit card) may be used. But just as push systems completely reverse the pull model, they also change the traditional payment assumptions. The sender may be interested in charging for all the data it sends out, especially since the receiver has subscribed to the information explicitly, but the receiver is only interested in paying for what is actually read (micro-payments, pay-per-view).
Figure 6.11 shows a simplified scheme for a Minstrel interaction if no payment is involved.
The broadcaster sends an offer (actually a sample including an offer) to the receiver (step 1). If the receiver is interested it requests the offered content (step 2) which the broadcaster then delivers (step 3).

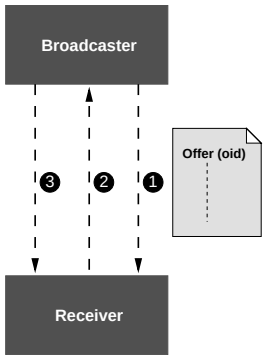

Figure 6.11: Minstrel interaction without payment

When payment is integrated into this communication model the interaction becomes more complicated. Figure 6.12 depicts the general payment model applied in Minstrel.

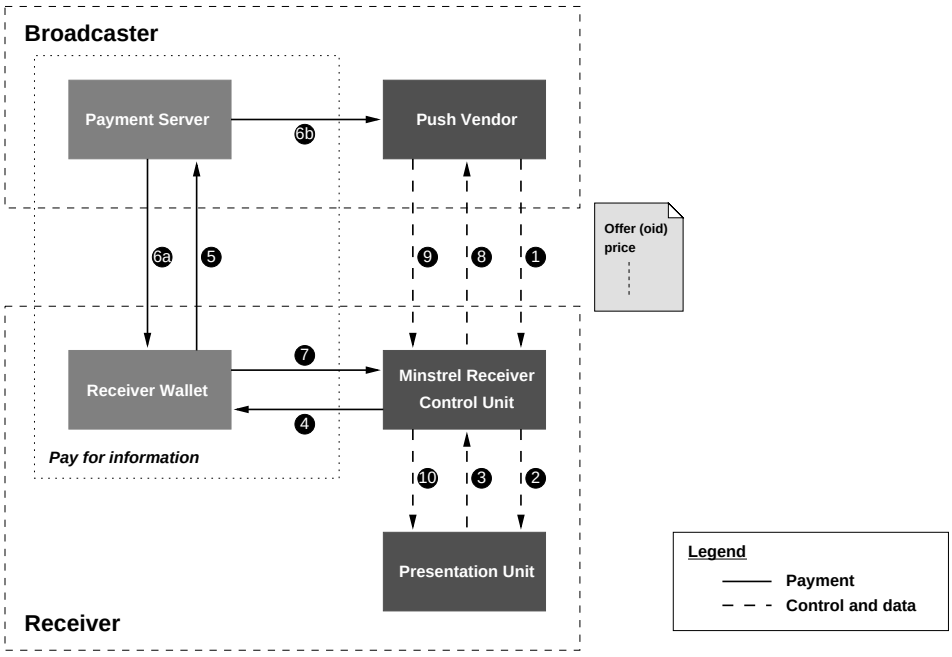

Figure 6.12: Minstrel interaction with payment

Again the broadcaster offers some information to the receiver (steps 1–2). But here the offer also includes a price and administrative information on how the payment can be effected. If the user is willing to accept—this can include user interaction or be done semi-automatically depending on the price and the offer—the following steps are taken. The user issues a request for the offered information which includes a payment handle. The payment handle identifies the offer and in turn the requested product (e.g., by including the unique ID of the offer), and holds

some administrative information (taken from the offer) that describes the payment process (step 3).
This handle is given to the user's wallet and the wallet is instructed to pay (step 4). On the basis of the payment process description included in the request, the wallet contacts a payment server and effects the payment according to the requested payment scheme/protocol (step 5). If the payment succeeds, the payment server issues a receipt that confirms the payment and is a proof of the transaction. This receipt is simultaneously sent to the wallet and registered with the push vendor (steps 6a–6b).
When the wallet gets the receipt, it returns it to the component that processes the user's request (step 7). Now the original request together with the receipt is sent to the push vendor (step 8). The vendor checks whether the receipt is valid, registered, and consistent with the requested product (i.e., the payment concerns the requested product). If this check is successful, the product is sent to the requester (step 9), which stores it and notifies the user of the completed operation (step 10).
Figure 6.13 depicts this payment model at the context level in terms of a UML sequence diagram.

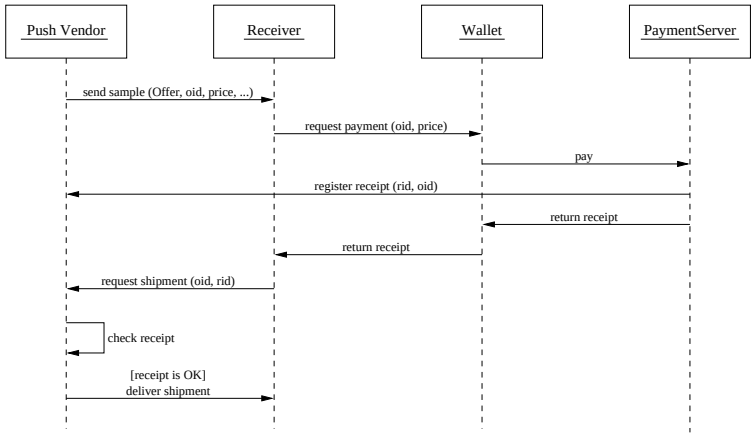

Figure 6.13: Minstrel interaction with payment at the context level (UML sequence diagram)

Figure 6.12 and Figure 6.13 show only the components that are relevant to the payment process. The transport system is not shown explicitly since it does not contribute to this process. The components of the transport system are transparent towards the payment process and merely have to forward the according requests.
Minstrel's payment model as described above is composed around the notion of a receipt, which makes it flexible and generally applicable for a variety of payment profiles. As presented, it can be mapped directly onto a *pay-per-view* business model using a micro-payment protocol, such as Millicent [54] or MicroMint [142]. However, the model is generic and can easily be tailored to other business models like time-based, volume-based, or flat fee schemes by adjusting or skipping part of the process defined above.
For example, a time-based scheme can be implemented by making an initial macro-payment for a specific vendor which would return a special (signed) receipt that also holds a time interval. If

the receiver (on behalf of the user) interacts with the vendor and is to be charged for that on a time basis, this is accomplished by sending the receipt with every user request for content. The vendor then checks whether there is time left on the receipt and, if so, decreases the time counter, registers the new receipt, invalidates the old one, and returns the changed receipt and requested information to the user. If the time counter is "empty," requests are denied and the user has to buy a new time ticket. This time-based scheme simply leaves out the individual payment with each access as shown in Figure 6.12.
Other business models can be implemented as well. If the counter is mapped to accesses instead of time, then it implements a pre-paid *n*-access scheme. If the receipt keeps track of the amount of data transferred, this models a volume-based scheme. A flat fee scheme would work in a similar way.
Moreover, micro-payment protocols and standards like DigiCash's ecash [34, 35, 36, 149] or CyberCash [27, 28], or payment standards like SET [151, 152, 153, 154], can be used under the abstraction of the Minstrel payment model by supplying feasible software components that conform to Minstrel's abstract payment interface which implements the payment model.
Due to the composition of the Minstrel payment model around the notion of a receipt, the business models, payment methods, and payment instruments do not exclude each other but can also co-exist and be used according to the user's configuration or requirements imposed by vendors.
As a proof of concept of the generic payment model described above, Minstrel includes an implementation of the Millicent [54] micro-payment protocol to support business models based on micro-payments such as pay-per-view.

6.2.1 Millicent Distilled

Minstrel uses the Millicent micro-payment protocol [54] to demonstrate a pay-per-view business model. Millicent is a micro-payment protocol developed by Digital Equipment Corporation (DEC) and is designed to support electronic payment on the Internet. Millicent is intended for small purchases (several cents to a few dollars), for example, to pay for a WWW page or buy a piece of information from a push channel.
This section provides an overview of Millicent and its main concepts. Further discussion, analysis, and details can be found in [54], [71], and [138].
Millicent is designed around the following concepts and roles: scrip (digital cash), broker, customer, and vendor. *Scrip* models an account a customer has established with a vendor or a broker. It represents digital money that is valid only for a specific vendor or broker and can be spent only once. The account balance is encoded in the scrip itself together with a proof of correctness for that value (digital signature) to prevent anyone from modifying the scrip's value.
A *broker* handles real-money transactions with vendors and customers and serves as an accounting intermediary between them. It takes care of account management, billing, and connection maintenance. Brokers buy and sell broker scrip and vendor scrip. Brokers establish long-standing accounts with vendors.
A *customer* establishes an account with a broker. Using brokers instead of vendors supports the splitting of a customer-vendor account into two accounts: one between the customer and broker,

and another between the broker and the vendor. Instead of many separate accounts for every customer-vendor combination, each customer has only a few accounts with a couple of brokers.
A *vendor* sells products and accepts its vendor-specific scrip as payment. A vendor establishes long-standing accounts with just a few brokers. The vendor can locally validate the scrip it receives to prevent customer fraud.
Figure 6.14 shows how the above entities cooperate in the Millicent payment process.

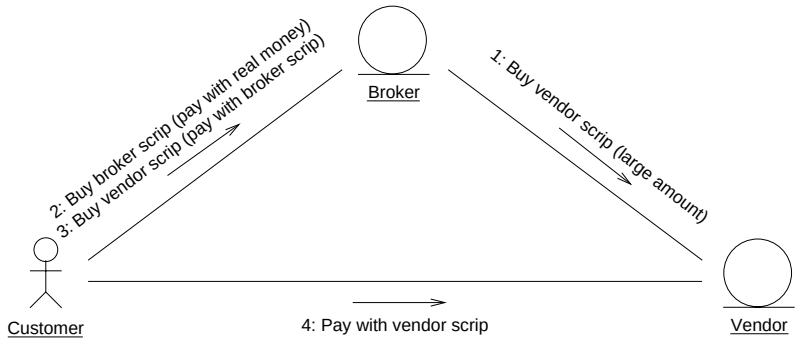

Figure 6.14: The Millicent payment model (UML collaboration diagram)

If a customer wants to purchase something from a specific vendor, it first checks whether it has enough vendor scrip (scrip for this specific vendor) to effect payment. If yes, it sends the vendor scrip to the vendor, the vendor locally checks the validity of the received scrip, deducts the cost of the purchase from the scrip's value, and returns new scrip with the new balance as change to the customer (4).
If the customer does not have enough vendor scrip, it first needs to buy the specific vendor scrip from the broker. Vendor scrip must be paid with broker scrip. If the customer does not have enough broker scrip, it first must buy the broker scrip from the broker with a real-money transaction which is outside the scope of Millicent. This payment is effected by some other payment method such as credit card or an electronic payment system with a higher-level of security (2). Broker scrip serves as a common currency for customers to use when buying vendor scrip, and for vendors to give as a refund for unspent scrip.
To be able to sell vendor scrip, the broker in an earlier step has established an account with every vendor it brokers for and bought a larger amount of vendor scrip from each vendor. This payment involves real money and is outside the scope of the Millicent protocol. It could be made with a credit card transaction or an electronic payment system with a higher-level of security (1).
Once the customer has broker scrip, it sends a request for the required vendor scrip to the broker together with some broker scrip to pay for it. The broker locally checks the validity of the received scrip, deducts the cost of the vendor scrip from the value of the broker scrip sent by the customer, and returns the requested vendor scrip and new broker scrip with the reduced balance to the customer (3). Now the customer can make the purchase as described above (4).
Micro-payment protocols generally trade security issues for lower transaction costs. This is important since the transaction costs must be considerably lower than the amount transferred

to be commercially reasonable. Since only small amounts of money are involved, "stealing" it would not pay off. Thus micro-payment protocols usually come with light-weight security compared to macro-payment protocols.
Millicent can employ different levels of security in its protocols depending on the intended application area. Different implementations of scrip serve as the basis for this. *Scrip in the clear* is the simplest but most efficient protocol. Scrip is sent without encryption or protection and an eavesdropping third party can intercept scrip being returned as change and use it. The amounts that would be lost, however, are rather small. An outside administrative process must then be run to identify the eavesdropping third party and sue it.
Private and secure employs a symmetric encryption method, such as DES [120] or IDEA [88], to setup a secure communications channel. This provides good privacy and protection against eavesdropping and theft but comes at additional cost.
Secure without encryption lies between these two extremes: it is secure but trades privacy for higher efficiency. It does not employ a full-blown encrypted channel but uses signed requests to prevent theft.

6.2.2 Using Millicent for Payment in Minstrel

The Millicent protocol is used in Minstrel in a concrete instantiation of the payment model of Figure 6.12. The model as shown in Figure 6.12, however, must be extended slightly to meet Millicent's special requirements. Figure 6.15 depicts the Millicent instantiation of the Minstrel payment model.

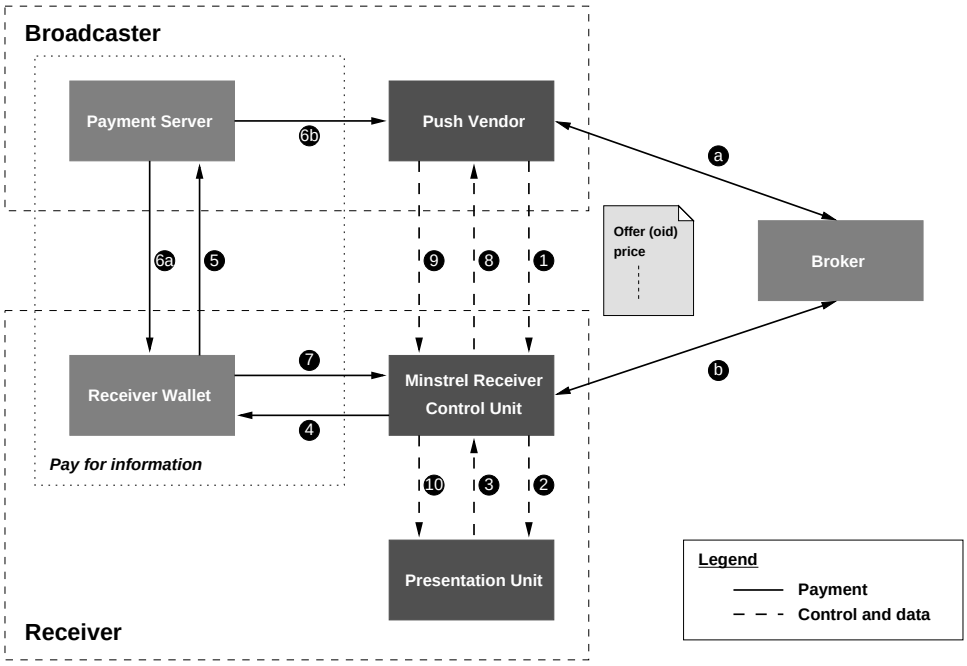

Figure 6.15: Minstrel interaction with payment using Millicent

In addition to the standard Minstrel payment model, the Millicent payment model includes brokers. Occasionally the broker buys vendor scrip from vendors (step a) and the receiver must buy broker scrip and vendor scrips from the broker (step b). These steps, however, occur infrequently and do not change the payment model itself. The other steps in the payment model remain unchanged.
The Millicent payment model shown in Figure 6.15 is included in Minstrel as a proof-of-concept implementation. The Millicent protocol is used for the realization of a pay-per-view business model. A detailed description of the implementation is given in [138]. Figure 6.16 shows the UML sequence diagram of a payment in Minstrel using the Millicent protocol.

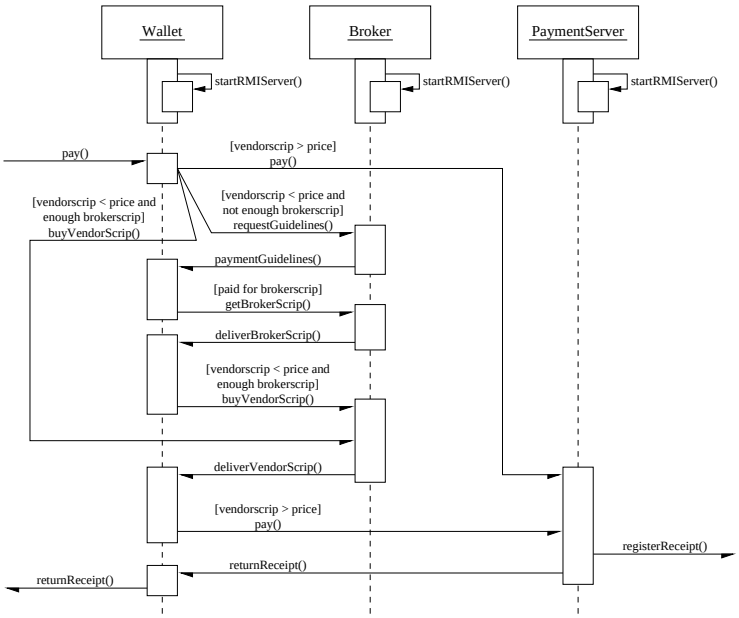

Figure 6.16: Pay-per-view payment using Millicent (UML sequence diagram)

All components in Figure 6.16 are accessible via RMI, so they initialize their RMI server components when starting. After this startup phase they are ready to serve payment requests. When the user issues a payment request, this request is forwarded to the wallet. The request includes all administrative information necessary to effect the requested payment. The wallet holds the user's scrips, keeps track of all on-going payment operations and provides status information on them. Having received a payment request, the wallet first checks whether the user has enough vendor scrip for the specific vendor to commence the payment.
If not, two situations can occur: The user either does or does not have enough broker scrip to buy vendor scrip from his/her broker for the payment. In the second case the wallet requests guidelines for buying broker scrip from the broker. This step is necessary since this interaction involves real money and is outside the scope of the Millicent protocol. The Millicent implementation in Minstrel expects an appropriate guideline object from the broker that allows the wallet to pay for the broker scrip using a higher-security electronic payment system (e.g., SET). On the basis of these guidelines the wallet buys broker scrip (possibly after an additional query to

the user whether s/he agrees to this payment). The modeling of payments outside Millicent as guideline objects facilitates abstraction from the concrete payment instrument used.
If the wallet has enough broker scrip or a suitable amount has been purchased as described above, it buys vendor scrip from the broker in order to accomplish the original payment request by the user. For this purpose the wallet sends broker scrip to the broker. The broker locally checks the validity of the received scrip, deducts the cost of the vendor scrip from the value of the broker scrip sent by the wallet, and returns the requested vendor scrip and new broker scrip with the reduced balance to the wallet. Now the wallet proceeds with paying the vendor.
For paying the vendor the wallet sends an appropriate amount of vendor scrip and administrative data to identify the purchase to the payment server. The payment server either is the vendor (in Millicent terminology) or acts on behalf of it, i.e., is a specialized component that offers a payment service to a number of vendors. Upon receipt of the request the payment server locally checks the validity of the received scrip and deducts the cost of the purchase from the scrip's value. Then it registers a receipt describing the purchase and the buyer with the (push) vendor. The receipt is also returned to the receiver as a proof of purchase together with the user's change in the form of new scrip with a reduced balance.
Now the wallet can notify the component of the Minstrel receiver that forwarded the original payment request and return the received receipt. The purchased shipment can now be requested from the vendor by including the receipt as a proof of payment in the request. The vendor checks the validity of the receipt and, upon positive evaluation, returns the requested shipment.
It is important to note that *all* interactions in the Millicent payment process shown in Figure 6.16 are done asynchronously, which provides a very high degree of parallelism. This requirement stems from the special nature of the payment process. As with graphical user interfaces the user can asynchronously issue requests and wants to continue working while the system tries to fulfill the request. For payment this means that the user will not want to wait until a payment transaction has been completed when s/he in the meantime can interact with Minstrel in a meaningful way, for example, by checking other channels. Moreover, the user may want to initiate a number of simultaneous payment operations. Thus the Millicent implementation in Minstrel is fully asynchronous and supports these requirements. Nevertheless, the user can monitor the status of ongoing payment processes at any time. After a payment succeeds or fails, the user receives a notification and can proceed as desired (depending on the result).
The above argument for asynchronous operations is also applicable to interactions between the internal components. The wallet, for example, should not be blocked, when waiting for the processing of a payment by a broker or a payment server. This goal could be accomplished by generating a new thread for each request, but that would not be very economical in terms of resources when a client carries out many parallel payment operations. With the asynchronous scheduling scheme of Minstrel's Millicent implementation, it is possible to "re-use" threads that cannot proceed because they are waiting for an answer from another party. This approach is similar to process or thread scheduling of operating systems and provides a very high degree of parallelism and efficient resource usage. Since a high payment load is envisioned for Minstrel with the further development of e-commerce, this approach was chosen.
The implementation uses a thread pool based on the *util.concurrent* library [92] to limit the maximum number of active threads. Minstrel's Millicent implementation also has to tackle or-

ganizational problems inherent to the Millicent approach. An important issue is how the total amount of scrip for a specific vendor is divided into smaller scrips. This is important because it directly influences the possible number of concurrent payments. If only one piece of scrip were available, no concurrent payments would be possible since the wallet would have to wait for the completion of the payment involving the scrip (it actually has to wait for the change scrip). This constraint is due to the fact that Millicent is based not on the concept of coins but on accounts: Scrip holds the value that it represents like an account. A simple approach to remedy this is to buy new vendor scrip whenever not enough is available. However, this strategy will possibly result in many scrips for the same vendor and leave the user with a very high account for every single vendor. This strategy is comparable to paying for every real-live purchase with a new high-value bill, without using any of the bills received as change. Additionally it is possible that the user has the amount to pay for a purchase but not as a single piece of vendor scrip. In real life this is not a problem, since the bills or coins used for payment do not influence the payment itself as long as the total amount is high enough. For Millicent, however, this is a problem since only one piece of scrip can be used for each purchase. Therefore the vendor scrip has to be cashed in and converted to a higher-value scrip. In answer to these and similar problems, Minstrel's Millicent implementation includes a complex model to support convenient strategies for concurrent payments. For example, in Minstrel the user can effect a payment with a number of scrips whose sum is the required amount.

A comprehensive description of Minstrel's Millicent implementation that provides further details is given in [138].

Chapter 7

Evaluation and Future Work

Even though there are many documents on the world-wide web and in electronic magazines about push systems, these are mostly at the user and application level, with little systematic treatment of the design and research issues. This thesis has presented push systems as an architectural model for distributed systems and interactions and has positioned it with respect to client-server and event-based architectures. The subscription phase of the interaction model is the key to the scalability of the push model and is applicable to many distributed applications for which client-server computing is deficient. A component model for push systems has been presented that can be used to study, analyze, and contrast different implementations of push systems, and this has been done for six prominent push systems. Using the concepts of broadcaster, transport system, receiver, and information source, the described component model separates the issues of content management, channel management, scalability, and user-interface management into different components. This component model may be used as a basis for a reference implementation of push systems.

Push systems have been contrasted with the closely related paradigm of event-based systems, the distinguishing features have been pointed out, and the connection with mobile code systems have been shown. Also the main issues that need to be addressed by push systems have been presented: scalability, network traffic, security, authentication, and electronic commerce. The Minstrel project covers all these issues. It uses the component model of Section 2.2 as an architecture for developing plug-compatible components for push systems and to devise an open protocol suite for Internet-scale content distribution.

Minstrel is a Java-based proof-of-concept implementation of the architectural model and serves as an extensible software platform for further research. The main design goals of Minstrel are scalability to a large number of users with simultaneous minimization of network traffic, a hybrid broadcasting paradigm that supports timely notification via real push without requiring special multicast infrastructures, a distributed model for simplifying information authentication, integrated support for the implementation of common payment schemes, and support for pushlets that are executed in a highly configurable Java secure execution framework (additive and subtractive security policies, hierarchical user groups, security negotiation). The protocols deployed are based on RMI and open to the public and can serve as a basis for discussion of a push protocol standard.

7.1 Evaluation

In the currently dominant "pull" paradigm on the Internet, consumers actively "pull" information from an information source. This imposes a "synchronous" interaction scheme that requires consumers to check for new information repeatedly. The "push" model attempts to overcome the deficiencies of the "pull" model by putting the information provider in control of the data flow and allowing it to actively transport information closer to its consumers. This provides "asynchronous" information distribution: Whenever information of the consumer's choice becomes available it gets distributed. In the "push" model producers announce the availability of certain types of information, an interested consumer subscribes to this information, and the producer publishes the new information whenever it becomes available (pushes it to the consumer).
The main challenges a push system faces are:

- Scalability
  - Timeliness of information despite large numbers of consumers
  - Reasonable bandwidth requirements in the presence of large numbers of consumers
  - Limited computational load for the distribution
- Active, asynchronous information distribution to meet the requirement of information freshness
- Flexible support for a wide range of content types including mobile code
- Security
  - Authenticity and integrity of information
  - Confidentiality of information
  - Mobile code security
- Support for e-commerce with flexible support of various payment methods and business models

The Minstrel push systems addresses all these issues. It is based on the component and communication model for push systems presented in Chapter 2.
Scalability is supported by clearly separating producers and consumers by an intermediate transport system. The transport system is hierarchical, consists of repeaters, caches, and proxies and is transparent towards producers, consumers, and channels. Because of Minstrel's transport system the maximum distribution delay grows logarithmically with an order of $O(f * l + f) = O(f * (l + 1)) = O(f * \log_f(n))$.[1] This is considerably lower than for the serial distribution case

[1] Without loss of generality it is assumed that every component in every layer in the transport system has a fan-out of f, recipients are equally distributed among the disseminating components, and l layers are used. Thus the total number of receivers is $n = f^{l+1}$. Additionally it is assumed that all network connections have the same bandwidth. This setup is similar to the one shown in Figure 4.7.

(e.g., email) where the maximum delay grows with an order of $O(n)$. Minstrel's hybrid broadcasting approach further supports scalability by actively distributing small notification messages (push part) that allow the receiver to decide whether it is interested in the announced information and will retrieve it (pull part). In business terms this strategy could be termed as *promotion through product samples*. This approach distributes the dissemination load, avoids unnecessary deliveries, and cuts down on bandwidth consumption and computational load. Additionally, it meets the goal of active information distribution to support information freshness.
The types of content transported by Minstrel channels are not constrained in any respect. By default the standard web content types are supported. Additional or user-defined content types can easily be integrated by means of Minstrel's concept of agents, which provide freely customizable content handling facilities.
Authenticity and integrity of information is supported by Minstrel's distributed authentication infrastructure (Minstrel Data Lock). All system data and content information is signed by the sender and is authenticated by the receiver before delivery. For provision of confidentiality Minstrel does not offer a specialized component but can rely on industry-standard infrastructures such as TLS [33] (SSL [128]) which can easily be integrated. The design decision to base on existing standards already in widespread use was taken deliberately.
Since mobile code is an essential and first-class content type in Minstrel (agents, pushlets), a flexible and highly configurable secure execution environment is included. The Java Secure Execution Framework (JSEF) goes beyond Java's standard security model and supports additive and subtractive security policies, local and global security policies, and (interactive) runtime security negotiation. XML is used for the definition of the policies.
For the implementation of payment methods and business models, Minstrel offers a flexible and generic payment model that can be used for a variety of business models, such as pay-per-view, volume-based, or flat fee. It is composed around the concept of a receipt which allows the employed business model to be decoupled from the underlying payment method(s). No constraints on the semantics of a receipt and the supported business model exist.
In the following sections I will expand on these arguments.

7.1.1 Scalability of the Broadcasting Process

In assessing and evaluating the scalability of the broadcasting process, the first question was, which evaluation strategy to choose. Available strategies fall into three main categories: case studies, simulation, and analytical approaches.
Case studies provide reasonable results for a defined test environment. A set of typical configurations can be defined and evaluated. This approach, however, has some deficiencies for evaluating Internet-scale systems such as a push system: The case studies must be of reasonable size and components must be distributed over a large number of nodes with a variety of different network connections in order to yield plausible results. For a push system such as Minstrel this would mean a setup with several thousand users that are geographically dispersed and are connected to the Internet with representative network connections. Such a setting is very difficult to achieve and still leaves open the question whether it is representative. The experience with case studies

of similar size and environment has shown that it is frequently not possible to generalize the results [10, 18, 20, 86, 93, 134].
In such a situation the usual alternative is simulation. But simulation is not effective either, because the results cannot be realistic. For example, no authoritative standards for simulating user or application behavior exist and it is difficult to separate the computational components from the communication delays in a test environment that is only distributed over a limited number of nodes. Moreover, simulations can hardly take into account all variables that affect the simulated system in such a highly dynamic environment as the Internet, where hundreds of possible influences exist and can change considerably over time in an arbitrary way. Two representative examples that can have major impact on a system running on the Internet and are difficult to simulate are Domain Name Service (DNS) lookups and networks disconnects. On the one hand, a slow or overloaded DNS server can dramatically decrease the performance and scalability of any Internet-scale system. On the other, network disconnects can have significant impact on the queuing system of a push system (similar to electronic mail) and introduce considerable computing loads that affect the overall system performance.
This leaves us with the analytical approach, which is also not ideal but at least feasible. If the assumptions are reduced to measurable quantities, the analytical approach provides reasonable estimates. Additionally, the assumptions can be changed easily to include further experience which supports an iterative approximation of the behavior of a system. The results can then be compared to related systems to support assessment of the system under consideration.
For evaluating the scalability and characteristics of the broadcasting process of Minstrel, the analytic approach was chosen. It takes into account bandwidth to provide an approximation of the actual system behavior. Other factors such as processing load on the nodes, delays introduced by the processing and dispatching of messages, and the queuing delays are not taken into account. This, however, seems justified since bandwidth is still the resource with such a significant impact that other factors have only minor influence. Moreover, the other factors have less impact on Minstrel's broadcasting process than they have on comparable systems such as electronic mail or the world-wide web, since Minstrel has a distributed dissemination infrastructure that distributes the broadcasting load. Thus the figures provided in the evaluation of Minstrel's broadcasting process can be viewed as upper limits.
The key design issues that support the scalability of Minstrel's broadcasting process are the use of small notification messages and the hierarchical transport system. In Minstrel only a limited number of recipients is connected to one feeding node. If this number grows over a certain threshold then an additional layer is introduced and the recipients are distributed among a number of nodes. This builds up a directed acyclic graph. Without loss of generality a tree structure for the transport system can be assumed. As was shown in Section 4.6 this has major impact on the distribution delay of a message. With electronic mail the delays grow linearly according to the number of receivers of a message with an order of $O(n)$ while in Minstrel the delays grow logarithmically with an order of $O(f * \log_f(n))$ as described above.
As was shown in Section 4.6 the worst-case delay in Minstrel for an example configuration of 10,000 receivers would be only 1% of the worst-case serial distribution delay of electronic mail. Email is a good candidate for comparison since it is the most widespread system with a similar dissemination strategy. For the example configuration Minstrel has an average delay of 12.87

seconds and the median of the delays is 9.48 seconds. 72% of the receivers have a delay of less than 20 seconds (50% of the worst-case delay) and 58% have a delay that is less than the average delay. After having processed 2019 of 10,000 recipients (approximately 20%), electronic mail is considerably slower than Minstrel, provided that the queuing strategies are similar. Other advantages of Minstrel's strategy are that no single point of failure exists and resources are used more efficiently since not only one machine's computing power and network connection are exploited.

The above figures shows that the active (push) part of the broadcasting strategy is efficient. Due to the structure of the Minstrel transport system and its components, the pull part is efficient as well. The content information that is described by the small messages distributed in the push part of the dissemination strategy can only be requested from the feeding component (the component in the hierarchy that pushed the description to the recipient). Since most of the transport system components are caches and repeaters (which also cache), this builds up an implicit caching infrastructure that is comparable to the caching infrastructure of the world-wide web. This satisfies the requirements of low bandwidth consumption and reasonably fast response times. Minstrel includes caching as a first-class concept.

Two problems were encountered in the process of implementing Minstrel's broadcasting strategy. The first is the currently limited performance of Java applications. Minstrel is implemented entirely in Java and the limited performance characteristics of Java have a severe impact on the performance of the broadcasting process. The second stems from the use of RMI-based distribution protocols. Although the push part of the distribution protocol has reasonable performance, the pull part falls short in this area. This is due to the very different sizes of objects that are transported. The push part uses small objects to support scalability while the actual content retrieved in the pull part can be quite large. This severely affects the performance of RMI, since objects that are to be transferred via RMI must be instantiated and available in memory before they can be used in an RMI call. Unlike socket connections RMI does not support a chunked read-and-transfer mode.

7.1.2 Content Selection, Content Types, and Executable Content

Minstrel as a push system does not have as powerful and expressive filtering capabilities as many event-based systems do. Nevertheless, Minstrel includes enough information in its "samples" (the small messages actively distributed to the recipients) to support receivers in selecting content of interest. On the basis of the information provided in a sample the receiver may request the announced content or discard it. Samples have two types of attributes: system-defined and user-defined. The system-defined attributes include a textual description, a version, validity information, content type, price, etc., that may already provide enough information for selection. Additionally, the user-defined attributes can be exploited to communicate arbitrary information (key-value pairs). The receiver must be able to interpret this information in a meaningful way, however.

The content types sent in samples or retrieved in "shipments" are not constrained in any way. By default, the standard web content types are supported. If other content types are to be distributed the provider of the information can supply a specialized agent that knows how to deal

with a specific content type and attach it to samples and shipments. Thus agents can dynamically extend the capabilities of receivers and render updates of the receiver software to support new content types unnecessary. Furthermore, the content itself can hold executable code with arbitrary functionality that is intended for execution at the receiver site (pushlet). This supports the transportation of arbitrary functionality inside a channel and can be used for a wide range of applications. In this respect the Minstrel push system resembles the functionality of a mobile agent system.

7.1.3 Security

Security in Minstrel is supported in two respects: authentication and mobile code. To guarantee authenticity and integrity of information, Minstrel includes a distributed, layered, high-level authentication infrastructure which provides authentication as a first-class concept. Authentication of information is of premier importance for push system users in high-confidence businesses, such as news agencies or financial information services.

Only so-called *GuaranteedObjects* are exchanged between Minstrel components, which ensure authentic and unchanged content. Both system information and information content are signed and can be verified by the receiver. The authentication infrastructure is component-based and layered to support separation of concerns in terms of specialized authentication servers and users of the infrastructure. The implementation already provides the necessary infrastructure and functionality for authentication purposes. However, it does not yet work with standard certification authorities and the robustness against brute-force security attacks has to be further investigated.

Mobile code is an important concept in Minstrel. Agents can dynamically extend receiver software with new capabilities and pushlets model arbitrary executable content sent in a channel. Both types of mobile code must be implemented in Java. Minstrel's mobile code security is based on Java's security model, which provides strong mechanisms to protect a system from security threats. Java's security model falls short, however, in terms of flexibility and security configuration management.

The Java Secure Execution Framework (JSEF) of Minstrel was built on top of the Java security model and provides powerful additional features. Java uses a security policy in which permitted accesses have to be explicitly stated. JSEF enhances this with the possibility to specify what is forbidden (subtractive policy). Java has a single-layer security model, while in JSEF security policies can be grouped and organized hierarchically. A global policy can be defined to state system-wide security standards. Users can tailor this policy towards their needs but cannot break it. The system-wide or so-called global policy overrules the user-defined local policies. Thus JSEF supports easier maintenance of mobile code security and helps to avoid the introduction of security holes by erroneous configurations. Security policies are specified using XML. Additionally, JSEF offers the possibility to negotiate security interactively at runtime. In the standard Java security model, a forbidden access typically terminates the execution; with JSEF, forbidden accesses are intercepted and the user (or a system component) can negotiate with the relevant Java code what accesses to permit.

The third aspect of security—confidentiality of information—is not supported by explicit means. Minstrel's design choice is to rely on components-off-the-shelf, that provide confidentiality by

means of encryption. Protocols like TLS [33] (SSL [128]) can be used transparently underneath Minstrel's infrastructure for this purpose.

7.1.4 Payment

Minstrel has integrated support for electronic payment and offers a generic payment model which is composed around the notion of a receipt. The semantics of concrete payment models are encapsulated in receipts. No constraints on the semantics of a receipt and the supported business models exist. The receipt concept supports the decoupling of the employed business models from the underlying payment methods. Thus Minstrel's payment model is not tied to specific payment methods and makes a variety of business models possible, such as pay-per-view, volume-based, or flat fee, but also more complex models including special offers or discount systems.
Before electronic "goods" that require payment are delivered in Minstrel, the payment model is executed. Regardless of the payment instrument used (micro-payment or macro-payment protocols) the buyer gets a receipt for an effected payment that can hold a description of the semantics of the payment. The vendor also gets this receipt and registers it. When the buyer then requests the goods from the vendor s/he encloses the receipt with the request. This allows the vendor to check the validity of the request before delivering the shipment. This model and the notion of a receipt are flexible and can be deployed for the implementation of various payment schemes depending on the semantics assigned to the receipt.
Besides the general framework, Minstrel includes a proof-of-concept implementation for a pay-per-view scheme based on the Millicent micro-payment protocol. The Millicent implementation was done as part of the Minstrel project.
Although Minstrel's payment model is generic and flexible, it requires some knowledge at the provider and the consumer to interpret the semantics of the employed business model. Support for the standard models is easy to provide, while complex models require more powerful and expressive mechanisms that are not currently included in Minstrel. An XML-based mechanism for automating the business logic between business partners is desirable and could be supported by a future version of Minstrel.

7.2 Future Work

The experiences in the course of the Minstrel project have pointed out several areas for future work and further investigation. The evaluation of the implementation of the Minstrel broadcasting strategy has shown that Java and RMI introduce overheads that require optimizations (outlined in Section 4.7.5). A new version of the distribution protocols will use a mixed RMI/socket implementation for the distribution of shipments to support a more efficient stream-oriented transfer mode. Additionally, the applicability of multicast RMI in MADP will be investigated. Besides these system-level optimizations, improvements at the user-level, such as more powerful personalizing und filtering capabilities, will be added.
Special focus will be put on extending Minstrel's e-commerce facilities. The current version does not include runtime negotiation of business model and payment method. For this, XML [14]

could be exploited to develop a definition language for business models and payment to facilitate dynamic configurations. This would improve Minstrel's usability as a platform for information commerce over the Internet. Furthermore, the addition of other micro-payment and macro-payment protocols is planned.

Finally, recent attempts at standardization in the areas of information dissemination and business relationships over the Internet, such as the Information and Content Exchange (ICE) protocol [174], will be evaluated as to whether their concepts and formats can be applied to or used in Minstrel.

Bibliography

[1] Y. Aahlad, B. Martin, M. Maratbe, and C. Le. Asynchronous notifications among distributed objects. *Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS)* (Toronto, Ontario, Canada). USENIX Association, June 1996.

[2] AltaVista Corporation. *AltaVista website*, 1999. http://www.altavista.com/.

[3] Apple Computer Incorporated. *QuickTime*, 1999. http://www.apple.com/quicktime/.

[4] K. Arnold and J. Gosling. *The Java programming language*. Addison-Wesley, Reading, Mass. and London, 1996.

[5] BackWeb Technologies. *BackWeb creative guide*, 1999. http://support.backweb.com/public/Version5.01/CREATIVE/INDEX.HTM.

[6] BackWeb Technologies. *BackWeb Polite Server*, 1999. http://support.backweb.com/public/Version5.01/SERVER/INDEX.HTM.

[7] BackWeb Technologies. *BackWeb—a cooperative architecture for a flexible push-pull broadcasting solution*, March 1997. http://www.backweb.com/pd/whitepaper.html.

[8] J. M. Bacon, J. Bates, R. J. Hayton, and K. Moody. Using events to build distributed applications. *Proceedings of the Second International Workshop on Services in Distributed Networked Environments*, pages 148–55. IEEE Computer Society Press, June 1995.

[9] R. A. Barta and M. Hauswirth. Interface-parasite gateways. *Fourth International World Wide Web Conference* (December 11–14, 1995, Boston, Massachusetts, USA). Published as *World Wide Web Journal*, **1**(1):277–90. O'Reilly & Associates, Incorporated, November 1995. http://www.infosys.tuwien.ac.at/Staff/pooh/papers/BIBOS/.

[10] V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, **12**(7):733–43, July 1986.

[11] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. Network Working Group, May 1996. RFC 1945. http://www.ietf.org/rfc/rfc1945.txt.

[12] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Second International World Wide Web Conference* (Chicago, USA, October 17–20, 1994). Published as I. Goldstein and J. Hardin, editors, *Computer Networks and ISDN Systems*, **28**(1&2):119–25. Elsevier Science B.V., December 1995.

[13] S. Brandt and A. Kristensen. Web push as an Internet notification service. Technical report. Hewlett-Packard Laboratories, Bristol, UK, 1997. http://keryxsoft.hpl.hp.com/doc/ins.html.

[14] T. Bray, J. Paoli, and C. M. Sperberg. Extensible Markup Language (XML) 1.0. World Wide Web Consortium (W3C), 10 February 1998. W3C Recommendation. http://www.w3.org/TR/1998/REC-xml-19980210.html.

[15] B. Calandra. So ya wanna be a pusher? *developer.com journal*, 29 July 1998. developer.com journal: staff picks, http://www.developer.com/news/staffpicks/072998_picks.html.

[16] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in Supporting Event-based Architectural Styles. *Proceedings of the third international workshop on software architecture*, pages 17–20. Association for Computing Machinery, 1998. http://www.acm.org/pubs/articles/proceedings/soft/288408/p17-carzaniga/p17-carzaniga.pdf.

[17] D. W. Chadwick. *Understanding X.500 – The Directory*. Chapman & Hall, 1996.

[18] S. R. Chalup, C. Hogan, G. Kulosa, B. McDonald, and B. Stansell. Drinking from the fire(walls) hose: another approach to very large mailing lists. *Proceedings of the Twelfth Systems Administration Conference (LISA '98)* (Boston, MA, December 6–11, 1998), pages 317–26. USENIX Association, December 1998.

[19] D. B. Chapman. Majordomo: how I manage 17 mailing lists without answering "-request" mail. *Proceedings of the Sixth Systems Administration Conference (LISA '92)* (Long Beach, CA, October 19–23, 1992), pages 135–44. USENIX Association, 1992.

[20] N. Christenson, D. Beckemeyer, and T. Baker. A scalable news architecture on a single spool. *;login:*, **22**(3):41–5, June 1997.

[21] B. Costales and E. Allman. *sendmail*. O'Reilly & Associates, Incorporated, January 1997.

[22] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems – concepts and design*, International Computer Science Series, 2nd edition. Addison-Wesley, Reading, Mass. and London, 1994.

[23] G. Coulouris, J. Dollimore, and T. Kindberg. Security. In *Distributed systems – concepts and design*, International Computer Science Series, pages 477–516, 2nd edition. Addison-Wesley, Reading, Mass. and London, 1994.

[24] D. H. Crocker. Standard for the format of ARPA Internet text messages, August 1982. RFC 822. http://www.ietf.org/rfc/rfc0822.txt.

[25] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. *Proceedings of the 20th International Conference on Software Engineering (ICSE 98)* (Kyoto, Japan), April 1998.

[26] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. Technical report. CEFRIEL, Politecnico di Milano, Via Fucini, 2, 20133 Milano, Italy, August 1998.

[27] CyberCash, Incorporated. *CyberCash Documentation*, 1999. http://www.cybercash.com/cybercash/merchants/support/doclib.html.

[28] CyberCash, Incorporated. *CyberCash website*, 1999. http://www.cybercash.com/.

[29] G. Dauphin. mMosaic: Yet Another Tool Bringing Multicast to the Web. *W3C Workshop "Real Time Multimedia and the Web" (RTMW '96)* (INRIA, Sophia Antipolis, France, October 24–25, 1996). World Wide Web Consortium (W3C), October 1996. http://www.w3.org/AudioVideo/9610_Workshop/paper05/paper05.html.

[30] M. Day, J. F. Patterson, and D. Mitchel. The Notification Service Transfer Protocol (NSTP): infrastructure for synchronous groupware. *Sixth International World Wide Web Conference* (Santa Clara, California, USA, April 6–11, 1997). Published as *Computer Networks and ISDN Systems*, **29**(8–13):905–15. Elsevier Science B.V., September 1997. http://www.lotus.com/research.

[31] T. Dean and W. Ottaway. Domain Security Services using S/MIME, November 1998. Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-smime-domsec-01.txt.

[32] M. Decina, E. Di Nitto, A. Fuggetta, V. Trecordi, and J. Wojtowicz. ORCHESTRA: a retailing infrastructure for network-wide services. Technical report. CEFRIEL, Politecnico di Milano, Via Fucini, 2, 20133 Milano, Italy, February 1998.

[33] T. Dierks and C. Allen. The TLS Protocol Version 1.0. Network Working Group, January 1999. RFC 2246. http://www.ietf.org/rfc/rfc2246.txt.

[34] DigiCash, Incorporated. *An introduction to how ecash works*, 1997. http://digicash.com/ecash/docs/Works(23G).pdf.

[35] DigiCash, Incorporated. *Setting up a shop to accept ecash*, 1997. http://digicash.com/ecash/docs/ShopSet(23G).pdf.

[36] DigiCash, Incorporated. *DigiCash website*, 1999. http://digicash.com/.

[37] S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, and L. Repka. S/MIME Version 2 Message Specification. Network Working Group, March 1998. RFC 2311. http://www.ietf.org/rfc/rfc2311.txt.

[38] S. Dusse, P. Hoffman, B. Ramsdell, and J. Weinstein. S/MIME Version 2 Certificate Handling. Network Working Group, March 1998. RFC 2312. http://www.ietf.org/rfc/rfc2312.txt.

[39] C. Ellerman. Channel Definition Format (CDF). World Wide Web Consortium (W3C), 10 March 1997. W3C Note. http://www.w3.org/TR/NOTE-CDFsubmit.html.

[40] H. Eriksson. MBone: The Multicast Backbone. *Communications of the ACM*, **37**:54–60, August 1994.

[41] Excite Incorporated. *Excite website*, 1999. http://www.excite.com/.

[42] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Network Working Group, January 1997. RFC 2068. http://www.ietf.org/rfc/rfc2068.txt.

[43] M. Fischer. *A component-based substrate for secure and authenticated communication*. Master's Thesis. Distributed Systems Group, Technical University of Vienna, Austria, 2000. To be published.

[44] M. Fischer and M. Hauswirth. Minstrel Security Architecture. Distributed Systems Group, Technical University of Vienna, Austria, 1999. http://www.infosys.tuwien.ac.at/Minstrel/Security/.

[45] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *Proceedings of the 1995 ACM SIGCOMM Conference* (Cambridge, MA, August 1995), pages 342–56, August 1995. ftp://ftp.ee.lbl.gov/papers/srm1.tech.ps.Z.

[46] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples. Network Working Group, November 1996. RFC 2049. http://www.ietf.org/rfc/rfc2049.txt.

[47] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. Network Working Group, November 1996. RFC 2045. http://www.ietf.org/rfc/rfc2045.txt.

[48] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. Network Working Group, November 1996. RFC 2046. http://www.ietf.org/rfc/rfc2046.txt.

[49] N. Freed, J. Klensin, and J. Postel. Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. Network Working Group, November 1996. RFC 2048. http://www.ietf.org/rfc/rfc2048.txt.

[50] S. Fritzinger and M. Mueller. Java security. Sun Microsystems, Incorporated, 1996. White Paper. http://java.sun.com/security/whitepaper.txt.

[51] FTP Software. *World-class push technology from FTP Software*, 1997. http://www.ftp.com/product/whitepapers/push.html.

[52] S. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, Incorporated, December 1994.

[53] C. Ghezzi and M. Jazayeri. Pure virtual functions for specification. In *Programming language concepts*, page 305, 3rd edition. John Wiley, New York, 1998.

[54] S. Glassman, M. Manasse, M. Abadi, P. Gauthier, and P. Sobalvarro. The Millicent Protocol for Inexpensive Electronic Commerce. *Fourth International World Wide Web Conference* (Boston, Massachusetts, USA). Published as *World Wide Web Journal*, **1**(1). O'Reilly & Associates, Incorporated, November 1995. http://www.w3.org/Conferences/WWW4/Papers/246/.

[55] L. Gong. Secure Java Class Loading. *IEEE Internet Computing*, **2**(6):56–61, November/December 1998.

[56] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: an overview of the new security features in the Java Development Kit 1.2. *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (Monterey, California, December 1997). USENIX Association, 1997.

[57] R. S. Gray. Agent Tcl: A transportable agent system. *Fourth International Conference on Information and Knowledge Management (CIKM 95)* (Baltimore, Maryland, December 1995), J. Mayfield and T. Finin, editors, November 1995.

[58] S. Gritzalis and D. Spinellis. Addressing threats and security issues in world wide web technology. *Proceedings of CMS'97, 3rd IFIP TC6/TC11 International Joint Working Conference on Communications and Multimedia Security* (Athens, Greece), pages 33–46, September 1997.

[59] T. Gschwind and M. Hauswirth. A Cache Architecture for Modernizing the Usenet Infrastructure. *32nd Hawaii International Conference on System Sciences (HICSS-32)* (Maui, Hawaii, USA, January 5–8, 1999), January 1999. http://www.infosys.tuwien.ac.at/Staff/pooh/papers/NewsCache/.

[60] T. Gschwind and M. Hauswirth. NewsCache – A High Performance Cache Implementation for Usenet News. *USENIX Annual Technical Conference* (Monterey, California, USA, June 6–11, 1999), June 1999. http://www.infosys.tuwien.ac.at/Staff/pooh/papers/NewsCacheHP/.

[61] R. Hackathorn. Publish or Perish. *BYTE*, **22**(9):65–72, September 1997.

[62] R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. *17th International*

Conference on Distributed Computing Systems (Baltimore, SA, May 1997), pages 269–78, May 1997. ftp://ftp.cs.colorado.edu/users/andre/papers/ICDCS97.ps.

[63] R. S. Hall, D. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, CA, USA, May 16–22, 1999), pages 174–83, May 1999.

[64] G. Hamilton, R. Cattell, and M. Fisher. *JDBC database access with Java: a tutorial and annotated reference*, The Java series. Addison-Wesley, Reading, Mass. and London, 1997.

[65] M. Hapner, R. Burridge, and R. Sharma. Java Message Service. Sun Microsystems, Incorporated, 5 October 1998. http://java.sun.com/products/jms/jms-101-spec.pdf.

[66] E. R. Harold. Remote Method Invocation. In *Java Network Programming*, The Java Series, pages 347–74. O'Reilly & Associates, Incorporated, February 1997.

[67] M. Hauswirth. The Minstrel Push System Project website. Distributed Systems Group, Technical University of Vienna, 1999. http://www.infosys.tuwien.ac.at/Minstrel/.

[68] M. Hauswirth and S. Jakl. Netscape Remote Control Facility. Distributed Systems Group, Technical University of Vienna, Austria, 1999. http://www.infosys.tuwien.ac.at/Minstrel/Receiver/NRCF/.

[69] M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. *Proceedings of the ESEC/FSE 99 – Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)* (Toulouse, France, September 6–10, 1999), September 1999. http://www.infosys.tuwien.ac.at/Staff/pooh/papers/PushIssues/.

[70] M. Hauswirth, C. Kerer, and R. Kurmanowytsch. Minstrel Client Security Framework. Distributed Systems Group, Technical University of Vienna, Austria, 1999. http://www.infosys.tuwien.ac.at/Minstrel/Receiver/CSF/.

[71] M. Hauswirth and M. Pührerfellner. Minstrel E-Commerce. Distributed Systems Group, Technical University of Vienna, Austria, 1999. http://www.infosys.tuwien.ac.at/Minstrel/E-Commerce/ and http://www.infosys.tuwien.ac.at/Minstrel/E-Commerce/Millicent/.

[72] M. Hebert. *A push in the web direction*. The MITRE Corporation, July 1997. Common Datacast Architecture (CDA). http://www.mitre.org/pubs/edge/july_97/fourth.htm.

[73] P. Hoffman. Examples of CMS Message Bodies, February 1999. Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-smime-examples-00.txt.

[74] P. Hoffman. Enhanced Security Services for S/MIME, March 1999. Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-smime-ess-12.txt.

[75] C. Honton. Service Discovery Protocol, December 1997. Internet Draft. http://www.ietf.org/internet-drafts/draft-honton-sdp-02.txt.

[76] T. J. Hudson and E. A. Young. SSLeay and SSLapps FAQ, September 1998. http://www.psy.uq.edu.au:8080/~ftp/Crypto/.

[77] Institute for Applied Information Processing and Communications (IAIK). *iSaSiLk Toolkit*, 1999. http://jcewww.iaik.tu-graz.ac.at/iSaSiLk/iSaSiLk.htm.

[78] Intermind Corporation. *About Intermind's Communications Patents*, 1999. http://www.intermind.com/materials/patent_desc.html.

[79] International Telecommunication Union (ITU). *Telecommunication Standardization Sector (ITU-T), Information Technology – Open Systems Interconnection – The Directory: Authentication Framework*, June 1997. ITU-T Recommendation X.509.

[80] JCP Computer Services. *JCP SSL-Pro*, 1999. http://www.jcp.co.uk/secProduct/security_ssl_index.htm.

[81] M. Jensen. BLIP Protocol – Draft 0.006. BLIP.org, 10 August 1998. http://www.blip.org/protocol.htm.

[82] B. Kantor and P. Lapsley. Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News. Network Working Group, February 1986. RFC 977. http://www.ietf.org/rfc/rfc0977.txt.

[83] G. Karjoth, D. B. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, **1**(4), July 1997. http://computer.org/internet/ic1997/w4068abs.htm.

[84] C. Kerer. *A flexible and extensible security framework for Java code*. Master's Thesis. Distributed Systems Group, Technical University of Vienna, Austria, October 1999.

[85] J. Knudsen. *Java Cryptography*, The Java Series. O'Reilly & Associates, Incorporated, 1998.

[86] R. Kolstad. Tuning sendmail for large mailing lists. *Proceedings of the Eleventh Systems Administration Conference (LISA '97)* (San Diego, California, October 1997), pages 195–204. USENIX Association, October 1997.

[87] V. Kumar. *MBone: Interactive Multimedia On The Internet*. Macmillan Publishing, November 1995.

[88] X. Lai. On the design and security of block ciphers. *ETH Series in Information Processing*, **1**. Hartung-Gorre Verlag, Konstanz, 1992. Institute for Signal and Information Processing, ETH Zentrum, Zürich, Switzerland.

[89] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. World Wide Web Consortium (W3C), 22 February 1999. W3C Recommendation. http://www.w3.org/TR/REC-rdf-syntax/.

[90] D. Lawrence and H. Spencer. *Managing USENET*. O'Reilly & Associates, Incorporated, January 1998.

[91] D. Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley, Reading, Mass. and London, 1997.

[92] D. Lea. Overview of package util.concurrent, 1999. http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[93] A. S. Lee. A Scientific Methodology for MIS Case Studies. *MIS Quaterly*, pages 33–50, March 1989.

[94] T. Liao. Light-weight Reliable Multicast Protocol Specification, 13 October 1998. Internet Draft. http://www.ietf.org/internet-drafts/draft-liao-lrmp-00.txt.

[95] T. Liao. WebCanal White Paper. INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, 31 December 1997. http://webcanal.inria.fr/white/index.html.

[96] T. Liao. WebCanal: a multicast web application. *Sixth International World Wide Web Conference* (Santa Clara, California, USA, April 6–11, 1997). Published as *Computer Networks and ISDN Systems*, **29**(8–13):1091–102. Elsevier Science B.V., September 1997. http://webcanal.inria.fr/webcanal/www6.html.

[97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Mass. and London, 1997.

[98] Live Networks, Incorporated. *liveCaster*, 1999. http://www.live.com/liveCaster/.

[99] Live Networks, Incorporated. *LIVE.COM*, 1999. http://www.mbone.com/ or http://www.live.com/.

[100] Live Networks, Incorporated. *multikit*, 1999. http://www.live.com/multikit/.

[101] C. Low, J. Randell, and M. Wray. Self-Describing Data Representation (SDR), October 1997. Internet Draft. http://www.ietf.org/internet-drafts/draft-low-sdr-00.txt.

[102] W. Lugmayr. *Gypsy: A Component-oriented Mobile Agent System*. PhD thesis. Technical University of Vienna, December 1999. To be published.

[103] C. Ma and J. Bacon. COBEA: a CORBA-based event architecture. *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)* (Santa Fe, New Mexico, April 27–30, 1998). USENIX Association, April 1998.

[104] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: a push-based distribution substrate for Internet applications. *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (Monterey, California). USENIX Association, December 1997.

[105] Marimba, Incorporated. *The Castanet product family*, 1997. http://www.marimba.com/doc/general/current/introducing/introducing.html.

[106] Marimba, Incorporated. *Developing Castanet channels*, 1997. http://www.marimba.com/doc/Castanet_Developer_Docs/current/index.html.

[107] Marimba, Incorporated. *Introducing Castanet*, 1999. http://www.marimba.com/doc/40/intro/intro-castanet.fm.html.

[108] G. McGraw and E. Felten. Java security and type safety. *Byte*, **22**(1):63–4, January 1997.

[109] G. McGraw and E. W. Felten. *Java security: hostile applets, holes, and antidotes*. John Wiley, New York, 1997.

[110] G. McGraw and E. W. Felten. *Securing Java: getting down to business with mobile code*. John Wiley, New York, 1999.

[111] Microsoft Corporation. *Microsoft Internet Explorer*, 1999. http://www.microsoft.com/ie/.

[112] Microsoft Corporation. *Webcasting in Microsoft Internet Explorer 4.0 White Paper*, September 1997. http://www.microsoft.com/ie/press/whitepaper/pushwp.htm.

[113] K. Miller, K. Robertson, A. Tweedly, and M. White. StarBurst Multicast File Transfer Protocol (MFTP) Specification, April 1998. Internet Draft. http://www.ietf.org/internet-drafts/draft-miller-mftp-spec-03.txt.

[114] K. Moore. MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text. Network Working Group, November 1996. RFC 2047. http://www.ietf.org/rfc/rfc2047.txt.

[115] Mozilla Organization. *Mozilla.org website*, 1999. http://www.mozilla.org/.

[116] J. Myers and M. Rose. Post Office Protocol – Version 3. Network Working Group, May 1996. RFC 1939. http://www.ietf.org/rfc/rfc1939.txt.

[117] National Center for Supercomputing Applications, University of Illinois at Urbana Champaign. *webcast: collaborative document sharing via the MBone*, 1995. http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/webcast.html.

[118] National Center for Supercomputing Applications, University of Illinois at Urbana Champaign. *NCSA Mosaic Common Client Interface*, March 1995. http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/cci-spec.html.

[119] National Center for Supercomputing Applications, University of Illinois in Urbana-Champaign. *NCSA Mosaic for the X Window System*, January 1997. http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/.

[120] National Institute for Standards and Technology (NIST). *Data Encryption Standard (DES)*, December 1993. Federal Information Processing Standards Publication 46-2. http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[121] Netscape Communications Corporation. *An exploration of dynamic documents*, 1995. http://home.netscape.com/assist/net_sites/pushpull.html.

[122] Netscape Communications Corporation. *In-Box Direct*, 1999. http://home.netscape.com/ibd/.

[123] Netscape Communications Corporation. *My Netscape*, 1999. http://my.netscape.com/.

[124] Netscape Communications Corporation. *My Netscape Network: Quick Start Guide*, 1999. http://my.netscape.com/publish/help/mnn20/quickstart.html and http://my.netscape.com/publish/help/quickstart.html.

[125] Netscape Communications Corporation. *Netscape Communicator*, 1999. http://home.netscape.com/download/index.html.

[126] Netscape Communications Corporation. *Netscape Netcenter*, 1999. http://www.netscape.com/.

[127] Netscape Communications Corporation. *JavaScript Guide*, November 1997. http://developer.netscape.com/docs/manuals/communicator/jsguide4/index.htm.

[128] Netscape Communications Corporation. *Introduction to SSL*, October 1998. http://developer.netscape.com:80/docs/manuals/security/sslin/contents.htm.

[129] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, **9**(1):38–71, 1984.

[130] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1995. Revision 2.0.

[131] Object Management Group. *CORBAservices: Common Object Services Specification*, July 1997. formal/97-07-04.

[132] P. Parnes. RTP extension for Scalable Reliable Multicast, November 1996. http://www.cdt.luth.se/~peppar/docs/rtp_srm/draft-parnes-rtp-ext-srm-01.txt.

[133] P. Parnes, M. Mattsson, K. Synnes, and D. Schefström. The mWeb presentation framework. *Sixth International World Wide Web Conference*. Published as *Computer Networks and ISDN Systems*, **29**(8–13):1083–90. Elsevier Science B.V., September 1997.

[134] D. E. Perry, A. A. Porter, and L. G. Votta. A Primer on Empirical Studies. *Proceedings of the 19th International Conference on Software Engineering (ICSE 97)* (Boston, Massachusetts, USA, May 17–23, 1997), pages 657–8, May 1997.

[135] PointCast. *Product documentation*, 1999. http://www.pointcast.com/products/intranet/techresources/documentation.html?ibttechp.

[136] PointCast. *Technical papers*, 1999. http://www.pointcast.com/products/intranet/techresources/techp.html?ibtdoc.

[137] J. B. Postel. Simple Mail Transfer Protocol, August 1982. RFC 821. http://www.ietf.org/rfc/rfc0821.txt.

[138] M. Pührerfellner. *An implementation of the Millicent micro-payment protocol and its application in a pay-per-view business model*. Master's Thesis. Distributed Systems Group, Technical University of Vienna, Austria, December 1999. To be published.

[139] B. Ramsdell. S/MIME Version 3 Certificate Handling, April 1999. Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-smime-cert-08.txt.

[140] B. Ramsdell. S/MIME Version 3 Message Specification, April 1999. Internet Draft. http://www.ietf.org/internet-drafts/draft-ietf-smime-msg-08.txt.

[141] RealNetworks. *RealNetworks – The Home of Streaming Media*, 1999. http://www.real.com/.

[142] R. L. Rivest and A. Shamir. PayWord and MicroMint – Two Simple Micropayment Schemes. *CryptoBytes*, **2**(1):7–11. RSA Laboratories, 1996. http://theory.lcs.mit.edu/~rivest/RivestShamir-mpay.ps.

[143] P. Rodriguez and E. W. Biersack. Continuous multicast push of web documents over the Internet. *IEEE Network*, pages 18–31, March/April 1998.

[144] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. *Proceedings of the ESEC/FSE '97 – 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Zurich, Switzerland, September 22–25, 1997). Published as Mehdi Jazayeri and Helmut Schauer, editors, *Lecture Notes in Computer Science*, pages 344–60. Springer Verlag, Berlin, September 1997.

[145] D. S. Rosenblum, A. L. Wolf, and A. Carzaniga. Critical Considerations and Designs for Internet-Scale, Event-Based Compositional Architectures. *Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures* (Monterey, CA), January 1998.

[146] A. D. Rubin and D. E. Geer. Mobile Code Security. *IEEE Internet Computing*, **2**(6):30–4, November/December 1998.

[147] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*, Object Technology Series. Addison-Wesley, Reading, Mass. and London, 1999.

[148] B. Schneier. *Applied cryptography: protocols, algorithms and source code in C*, 2nd edition. John Wiley, New York, 1996.

[149] B. Schoenmakers. Basic security of the ecash payment system. *Computer Security and Industrial Cryptography: State of the Art and Evolution, ESAT Course* (Leuven, Belgium, June 3–6, 1997). Published as Bart Preneel, editor, *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1997. http://digicash.com/ecash/docs/cosic.pdf.

[150] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Network Working Group, January 1996. RFC 1889. http://www.ietf.org/rfc/rfc1889.txt.

[151] SET Secure Electronic Transaction LLC. *SET Secure Electronic Transaction Specification – Book 1: Business Description*, May 1997. Version 1.0. http://www.setco.org/download/set_bk1.pdf.

[152] SET Secure Electronic Transaction LLC. *SET Secure Electronic Transaction Specification – Book 2: Programmer's Guide*, May 1997. Version 1.0. http://www.setco.org/download/set_bk2.pdf.

[153] SET Secure Electronic Transaction LLC. *SET Secure Electronic Transaction Specification – Book 3: Formal Protocol Definition*, May 1997. Version 1.0. http://www.setco.org/download/set_bk3.pdf.

[154] SET Secure Electronic Transaction LLC. *External Interface Guide to SET Secure Electronic Transaction*, September 1997. http://www.setco.org/download/set_eig.pdf.

[155] J. Siegel. *CORBA fundamentals and programming*. John Wiley, New York, 1996.

[156] Silverspan Corporation. *Web Transporter*, 1999. http://www.silverspan.com/wtoverview.html.

[157] Software Research Laboratory. *Reliable Multicast Protocol*, August 1997. http://research.ivv.nasa.gov/RMP/index.html.

[158] StarBurst Software. *StarBurst MFTP – An Efficient, Scalable Method for Distributing Information Using IP Multicast*, 1997. http://www.starburstcom.com/white.htm.

[159] Sun Microsystems, Incorporated. *Java Plug-in Product*, 1999. http://java.sun.com/products/plugin/.

[160] Sun Microsystems, Incorporated. *The JNDI Tutorial*, 1999. http://java.sun.com/products/jndi/tutorial/index.html.

[161] Sun Microsystems, Incorporated. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*, April 1999. http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html.

[162] Sun Microsystems, Incorporated. *Secure computing with Java: now and the future*, September 1998. White Paper. http://java.sun.com/marketing/collateral/security.html.

[163] Telecommunications Information Networking Architecture Consortium. *TINA Notification Service Description*, July 1996. telecom/96-07-02.

[164] TIBCO Corporation. *TIB/Rendezvous*, 1999. http://www.rv.tibco.com/whitepaper.html.

[165] J.-C. Touvet. MultiCast Mosaic, May 1996. http://WWW.edelweb.fr/EdelStuff/EdelContrib/jctmcm.html.

[166] M. Umlauft. *A data store JavaBean for the Minstrel push system*. Master's Thesis. Distributed Systems Group, Technical University of Vienna, Austria, 2000. To be published.

[167] S. R. van den Berg. procmail mail processing package, 29 October 1995. ftp://ftp.informatik.rwth-aachen.de/pub/packages/procmail/procmail.tar.gz.

[168] A. van der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf. Software release management. *Proceedings of the ESEC/FSE '97 – 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Zurich, Switzerland, September 22–25, 1997). Published as Mehdi Jazayeri and Helmut Schauer, editors, *Lecture Notes in Computer Science*, pages 159–75. Springer Verlag, Berlin, September 1997.

[169] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. The HTTP Distribution and Replication Protocol. World Wide Web Consortium (W3C), 25 August 1997. W3C Note. http://www.w3.org/TR/NOTE-drp-19970825.html.

[170] A. van Hoff, H. Partovi, and T. Thai. The Open Software Description Format (OSD). World Wide Web Consortium (W3C), 13 August 1997. W3C Note. http://www.w3.org/TR/NOTE-OSD.html.

[171] G. Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis. Politecnico di Milano, Italy, 1997.

[172] Vitria Technology, Incorporated. *BusinessWare*, 1999. http://www.vitria.com/.

[173] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). Network Working Group, December 1997. RFC 2251. http://www.ietf.org/rfc/rfc2251.txt.

[174] N. Webber, C. O'Connell, B. Hunt, R. Levine, L. Popkin, and G. Larose. The Information and Content Exchange (ICE) Protocol. World Wide Web Consortium (W3C), 26 October 1998. W3C Note. http://www.w3.org/TR/1998/NOTE-ice-19981026.

[175] M. Weiss, A. Johnson, and J. Kiniry. Distributed Computing: Java, CORBA and DCE. Open Software Foundation Research Institute, February 1996. Version 1.2. http://www.informatik.uni-essen.de/Dokumente/java/corba.htm.

[176] D. Wessels and K. Claffy. ICP and the Squid Web Cache. *IEEE Journal on Selected Areas in Communication*, **16**(3):345–57, April 1998. http://ircache.nlanr.net/~wessels/Papers/icp-squid.ps.gz.

[177] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. *Theory and Practice in Distributed Systems, International Workshop* (Dagstuhl Castle, Germany, September 5–9, 1994). Published as K. P. Birman, F. Mattern, and A. Schiper, editors, *Lecture Notes in Computer Science*, **938**:33–57. Springer Verlag, Berlin, 1995. http://research.ivv.nasa.gov/RMP/Docs/RMP_dagstuhl.ps.

[178] J. E. White. Mobile Agents. In I. Bradshaw and M. Jeffrey, editors, *Software Agents*. MIT Press and American Association for Artificial Intelligence, 1997.

[179] WISEN: Workshop on Internet Scale Event Notification, July 13–14, 1998. Irvine Research Unit on Software (IRUS), Irvine (CA), USA, July 1998. http://www.ics.uci.edu/IRUS/wisen/.

[180] D. Reed and K. Jones, *Pushing push: advancing the features of channel communication*, *W3C Workshop on Push Technology* (Boston, USA, September 8-9, 1997). World Wide Web Consortium (W3C), September 1997. http://www.intermind.com/materials/pushing_push.doc.

[181] M. Wray and R. Hawkes. Distributed Virtual Environments and VRML: an event-based architecture. *Seventh International World Wide Web Conference* (April 14–18, 1998, Brisbane, Australia). Published as *Computer Networks*, **30**(1–7):43–51. Elsevier Science B.V., April 1998.

[182] Yahoo! Incorporated. *My Yahoo!*, 1999. http://my.yahoo.com/.

[183] Yahoo! Incorporated. *Yahoo! website*, 1999. http://www.yahoo.com/.

[184] R. J. Yarger, G. Reese, and T. King. *MySQL & mSQL*. O'Reilly & Associates, Incorporated, July 1999.

[185] F. Yellin. Low level security in Java. *Fourth International World Wide Web Conference* (Boston, Massachusetts, USA, December 11–14, 1995). Published as *World Wide Web Journal*, **1**(1). O'Reilly & Associates, Incorporated, November 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html.

Curriculum Vitae

Personal Details

February 8, 1966	Born in St. Johann/Pg., Salzburg, Austria
1972 – 1976	Primary school in St. Johann/Pg., Salzburg, Austria
1976 – 1984	Gymnasium (natural science track) in St. Johann/Pg., Salzburg, Austria. Graduated with distinction.
1984 – 1985	Study musicology and art history at University of Salzburg, Austria
1985 – 1986	Study computer science and psychology at University of Vienna, Austria
1986 – 1994	Study computer science at Technical University of Vienna, Austria (system software track) and worked part-time in industry. Graduated cum laude. Master thesis: *Companion – An object-oriented graphical user interface for TUNet services*
June 1995 – May 1996	Community service as an alternative to military service in a hospital.
March 1994 to date	Ph.D. student (Prof. Mehdi Jazayeri), teaching and research assistant at Distributed Systems Group, Technical University of Vienna, Austria.

Special Interests

- Push systems
- Java
- Mobile code
- E-commerce
- World-Wide Web
- Internet/Intranet
- Programming languages
- Distributed systems

Memberships

- Association for Computing Machinery (ACM)
- Institute of Electrical and Electronics Engineers (IEEE)

Publications

http://www.infosys.tuwien.ac.at/Staff/pooh/

XXXXXXXXXXXXXXXXXXXXXXXXXXXXX