A Component and Communication Model for Push Systems*

Manfred Hauswirth and Mehdi Jazayeri

Technical University of Vienna, Distributed Systems Group
{M.Hauswirth, M.Jazayeri}@infosys.tuwien.ac.at
http://www.infosys.tuwien.ac.at/

Abstract. We present a communication and component model for push systems. Surprisingly, despite the widespread use of many push services on the Internet, no such models exist. Our communication model contrasts push systems with client-server and event-based systems. Our component model provides a basis for comparison and evaluation of different push systems and their design alternatives. We compare several prominent push systems using our component model. The component model consists of producers and consumers, broadcasters and channels, and a transport system. We detail the concerns of each of these components. Finally, we discuss a number of open issues that challenge the widespread deployment of push or any other system on an Internet-wide scale. Payment models are the most important among these and are not adequately addressed by any existing system. We briefly present the payment approach in our Minstrel project.

1 Introduction

The dominant paradigm of communication on the worldwide 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. Even from the early days of the Internet, systems such as electronic mail and Usenet News have attempted to overcome the deficiencies of the pull model of communication by allowing producers of information to "push" their information closer to the clients. In the push model of communication, 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 Fig. 1.

The first push system approach was introduced in 1992 by the *dynamic document* concept of Netscape Navigator 1.1 [20]. 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 and a URL in the HTTP response or in the document header. After the given time the client loads the document specified by the URL.

* This work was supported in part by a grant from the Hewlett-Packard European Internet Initiative.

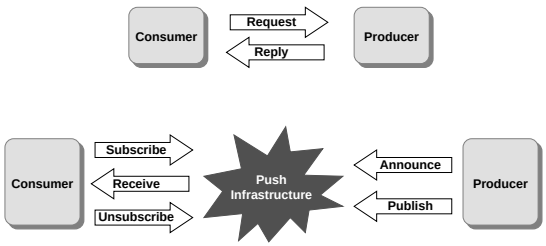

Fig. 1. Pull vs. push

To show the wide applicability of the push model as a paradigm for building distributed applications, here we outline three distinct applications whose design can be decomposed in terms of push concepts.

1. Intra-company employee information system. Many organizations have proprietary and ad hoc systems for keeping their employees informed about their organizational news. This is sometimes viewed as one of the organization's most important and most difficult tasks. Such a system may be built as a standard push system.
2. 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 maintenance manuals is costly and dealing with the paper manuals is tedious. With a push system each product line could be associated with one channel and each maintenance worker subscribes to the desired channel.
3. Stock ticker system. This is a classic example for event-based and push models.

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

The goal of this paper is to contrast the push communication model with existing communication models and present a component model for push systems. The component model provides a basis for comparison and evaluation of different push systems and their design alternatives. Further, the component model may be used as a basis for a reference implementation and as a source of identifying some open issues.

The paper is organized as follows: Section 2 compares the communication models for distributed systems and analyzes their impact on scalability, network load, and state maintenance. Section 3 presents our component model for push systems which is used in Sect. 4 for the comparison of representative push systems. Section 5 then discusses the relationship of push systems with mobile code and event-based systems. In Sect. 6 we present the main issues that push systems must address to become usable on an Internet scale. We summarize and give our conclusions in Sect. 7.

2 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 the distributed environment, the traditional software decomposition task 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 needed communication 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 state is shared between a client and a server and is modified through one or more interactions between them.

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. 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 client-server model deals with two participants in the communication. In the *peer-to-peer* model the application is decomposed among many peer nodes as opposed to clients and servers. For this reason, we refer to the nodes here 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, we may divide the peer-to-peer model 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 [28].

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

Table 1. Comparison of communication models

	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)

The four communication models of distributed systems described above occupy four areas in the design space of distributed systems. Figure 2 gives a rough view of the design space plotted along coupling and scalability axes.

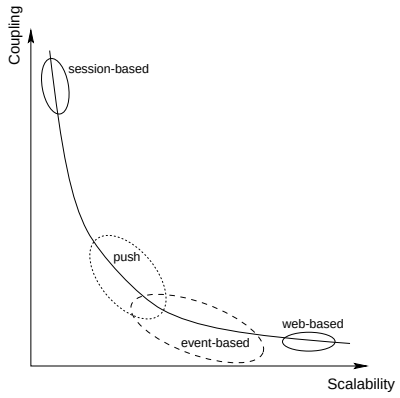

Fig. 2. Degree of coupling vs. degree of scalability

3 A Component Model for Push Systems

In this section, we present a component model for push systems which we have 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 is sent on the channel. A producer (*information source*) sends data through channels. Thus the connection phase of client-server systems is replaced by an early subscription phase.

In practice we use a broadcaster component to separate the concerns of channel handling from the information source. A broadcaster is responsible for managing channels and sending information along channels.

An information source feeds information into a broadcaster together with rules on how and where (which channel) to distribute this data. The broadcaster may apply filters to the data and then disseminate 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 the filters. To provide scalability to high numbers of users the distribution process involves a transport system which is conceptually transparent for broadcasters, receivers, and channels.

Figure 3 depicts our component model of a push system and Fig. 4 gives a sample collaboration (UML sequence diagram) among the components of the model.

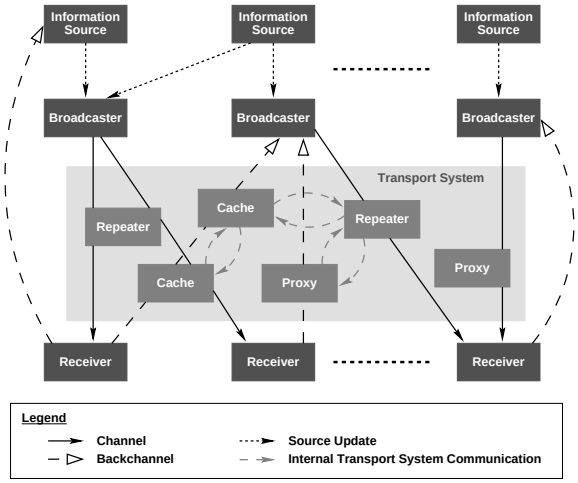

Fig. 3. Components of a push system

The information source provides new data for a specific channel to the broadcaster. The broadcaster applies filters on the data to limit data transfers and sends the data (in parallel or iteratively) to a set of repeaters (for scalability reasons) for which the filters succeeded. The repeaters then redistribute the data to 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 Fig. 3 some of the arcs representing channels and backchannels cut through components of the transport system to motivate that these components are necessary for scalability purposes but are transparent to the channels and the dissemination process.

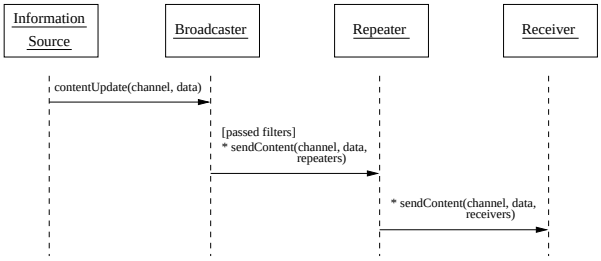

Fig. 4. Content distribution via a channel

Having decomposed a push system into the components in Fig. 3, we will now take a detailed look at the concerns of each component of the component model.

3.1 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 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. By grouping data according to the information type the total amount of data transfers can be easily cut down. Data (events) need only be distributed to channel subscribers. Additionally, finer filtering can be applied to the contents of a channel (like 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, software updates).

Data format: The formats (e.g. HTML, Java) and semantics (e.g. static, executable) of a channel's data.

Personalizing/filtering: 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 (possibly down to individual data pieces) must exist to prevent using up the consumer's resources.

Update strategy: This defines how updates of the channel's contents are done: replacement, incremental, or differential updates. 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. Content-scheduled channels deliver content "time in-

dependently."
Operation mode: Consumers may not be online all the time. Support for offline/mobile operation with feasible synchronization protocols is necessary.
Payment: Certain channels may involve payment (support channels, special contents, etc.). The channel configuration determines which payment scheme to use: pay-per-view, content-based, time-based, flat fee, etc.

Some of these properties can be modeled as (producer-side and consumer-side) data-stream (channel) filters. Filtering can be done both at the producer and at the consumer.

Channels model a 1:n relationship between a producer and its consumers. Additionally backchannels may facilitate "up-stream" communication as a 1:1 connector between a consumer and a producer. A consumer can communicate information back to a broadcaster or information source via a backchannel. This is usually done in a client-server style and thus is conceptually "outside" of push systems. 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 [27].

3.2 Broadcaster

A push system has at least one broadcaster component that offers channels and distributes channel 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 it cannot be a single component. For scalability, a specialized broadcasting infrastructure is necessary here.

The broadcaster itself may be distributed. A set of broadcasters may provide the channels and exchange updates among each other 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 [3].

The primary goal is that receivers 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.

3.3 Receiver

If we abstract from 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 human users and channels. It gets 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. According to 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 could query a channel directory or a specialized directory channel. Besides standard channels, specialized maintenance channels can exist that fulfill functions such as maintaining and

updating the push system's software components themselves, e.g., the receiver. 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 [8] and [9].

Push systems are related to mobile code systems since channels can distribute executable code. In analogy to applets we introduce the notion of a *pushlet*: executable code and data which is intended for execution at the receiver.

3.4 Transport System

In a large-scale setting, a dedicated transport system is necessary to make a push system scalable and operational, i.e. decrease network bandwidth consumption and increase 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. 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. 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" to the receiver.

Cache. A cache is the same as a repeater which is loaded dynamically rather than being preloaded (on-demand repeater).

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.

3.5 The Notion of Broadcasting

So far we have used the notion of broadcasting in the context of push systems only informally. Broadcasting in a large-scale push system cannot rely on a medium that offers broadcasting per-se (e.g. an Ethernet LAN). 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 [12]) and protocols (e.g. RTP [24], NSTP [5]). 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 induces high network traffic and a possibly high number of unnecessary messages if the channel data does not change frequently. On the other hand, consistency requirements of channels may be rather relaxed and pulling frequencies between 10 minutes and a day may suffice. Additionally, messages that are pulled may be rather small (some 100 bytes). Nevertheless, pulling is frequently used in push systems since it is robust, simple to implement, allows for off-line operation, and scales well to high numbers of subscribers.

Server push. The broadcaster actively sends content to its subscribed receivers. This solves the freshness problem of pulling but opens new problems. The main issue that appears in several ways is scalability: contacting the receivers sequentially does not scale even for medium numbers of subscribers. It would leave receivers with different views of channel information depending on their ordinal number in the pushing process. Server push broadcasting also requires a directory of subscribers to be contacted. That imposes additional administration since it must be maintained, kept consistent, and is a single point of failure. Additionally receivers may not be online all the time. This must be compensated by re-broadcasts which adds considerably to the broadcaster's load and complexity.

Hybrid approaches. Hybrid approaches combine the advantages of server push (freshness, consistency) and client pull (scalability): consumers are notified about the availability of new data via a push mechanism (small messages) while the client pulls to transfer the actual data (possibly large data). This approach is taken in the Minstrel project [19]: the broadcaster pushes a "sample" (description of 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 (like Netscape's Netcenter): users sign up with a mailing list and receive mails in regular intervals (push part); these mails typically hold a 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 and retrieve the corresponding document (pull part).

4 Representative Push Systems

Since 1996, a number of commercial systems have appeared that classify themselves as push systems. In this section we survey six prominent such systems. Table 4 compares the systems with respect to components and Table 4 classifies them in terms of the main features that were described in the previous sections. Providing such a comparison is surprisingly difficult due to the paucity of technical documentation on these systems. We were unable to find the answers to some of the entries in the tables. In the following, we briefly examine each system.

Table 2. Comparison of push systems based on the component model

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. Comparison of push systems based on features

Push System	Back-channel	Pushlets	Update Strategy	Filtering	Scalability	Receiver Update	Data Sec.
Castanet	plugin	√	diff. (file)	–	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	–	–

Castanet [16, 17] is an advanced push system for distributing content with specific emphasis on installing and updating software over the Internet. In Castanet the clients pull at configurable intervals down to 15 minutes, or if requested by the user. It supports HTML channels and pushlets (*presentation*, *applet*, or *application channels* written in Java). Updates are differential, i.e. only updated files in a channel are sent to the receivers. A limited backchannel functionality is provided by *plugins*: one plugin per channel allows to process feedback data, e.g. return language specific data based on the user's configuration. No explicit means for filtering exists but a limited degree is possible via user configurations. Castanet's transport system provides for high scalability: repeaters (*transmitters*; the broadcaster is called *primary transmitter*), caches (called *proxies*), and proxies (called *gateways*) that allow channel access behind firewalls. The information source is modeled by the *publisher* software component. The Castanet receiver (*tuner*) can be automatically updated. Castanet supports two security concepts: *channel signing* guarantees the integrity and authenticity of channel data and SSL provides encrypted transmission.

PointCast [21, 22] is both a push system and an information provider. Only content coming from registered information providers can be broadcast via the *Business Network*. The *Central Broadcast Facility* (CBF) is the central repository for PointCast network information. Additionally a freely configurable *intranet channel* for company information systems and *connections*—content from Web servers—are supported. Channel data consists of Web data formats and animations written in the *ScreenPlay* language, which can be considered a limited version of pushlets. CDF [6] can be used to

define (parts of) channels. The default pulling interval of clients is one hour (configurable by the user). Push distribution is available for intranets only (through multicast). PointCast has no broadcaster. The administrative and channel data are retrieved from Web servers. Several publishing tools exist. We have not found any explicit information on the update strategy. PointCast has no backchannel concept. Limited filtering is possible: users can select predefined content classes and types within channels. Only a limited set of channels can be subscribed in parallel. A cache (*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. Currently only a few CBFs exist. CBFs have limited support for load balancing (*PointRouter* and *PointServer*). Receiver software can be updated automatically. Distributed data is neither authenticated nor secured.

BackWeb [1, 2] 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* (BALI). It supports pull (HTTP and *BackWeb Transfer Protocol*) and push (based on StarBurst's *Multicast File Transfer Protocol*) distribution. Pull-based channels are queried every 5 minutes for updates by default (configurable). Pushlets can be executable files, Java applets and Netscape plugins. Differential updates are supported at a byte granularity and are transparent against network disconnects. Backchannels are available by the concept of *up-stream data* which supports the building of two-way push applications. Users can filter channel data by type and content. Scalability ranges from medium to high depending on the distribution protocols and transport infrastructure used. BackWeb's transport system uses chained caches (*proxy servers*). Receiver software has to be maintained manually. Certificates, digital signatures, and encryption are supported to ensure authenticated information and secure transmission.

Webcasting [18] is Microsoft's push technology. The receiver for channels is Internet Explorer (IE). Three types of Webcasting exist: *basic* ("crawling" a Web site), *managed* (a CDF [6] description defines the downloadable data—a list of URLs, its hierarchical structure, and an update schedule), and *"true"* (integration of third party software for multicast or other paradigms). Without third-party products the main paradigm of Webcasting is pull at user-configurable intervals. 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 content changes or downloading content changes and is notified of changes via IE or via email. No explicit backchannels exist, but they can be implemented "outside" by means of Java, ActiveX, DynamicHTML, etc. 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 fully relies on the Web infrastructure (Web servers, caches, etc.). Thus it is scalable to the degree the Web itself is. The receiver can be automatically updated via a special *software update channel*. Software update channels rely on OSD [26] which is a vocabulary to describe software components and their dependencies for deploy-

ment and is submitted to the W3C to become a standard. Software packages sent via a software update channel can be authenticated. Other data is neither authenticated nor secured.

WebCanal [14, 15] is a platform for *global information broadcast* on the Internet. It uses multicast push distribution based on the light-weight reliable multicast protocol [13] (an extension of RTP [24]) and the MBone [12]. It consists of a *WebCaster*, a *WebTuner*, and several other tools. WebCanal can also be used as a conferencing and presentation platform. Since it interacts with a web browser for content display the above statements for Webcasting concerning pushlets, filtering and backchannels apply here too though for backchannels it must be adjusted since WebCanal can also be used for symmetric two-way communication: every receiver can act as a broadcaster. Updates are differential at a file granularity. WebCanal relies on the MBone as its transport infrastructure. Due to this its scalability depends on MBone. Receiver software cannot be updated automatically though WebCanal supports software distribution channels. Distributed data is neither authenticated nor secured.

Intermind [27] is a pull-based push system. It has no 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 based on the executable content supported by the Web browser. Channels can be defined with the *channel publishing tool*. Such descriptions and the data are placed on a Web server where receivers can access it. We have not found any explicit information on the update strategy. It is also unclear how backchannels are supported. Backchannels could be implemented using features (Java, JavaScript, etc.) offered by the Web browser. While [27] defines a concept for data and meta-data exchange—*channel objects* based on XML and the Resource Description Framework (RDF)—between *communication nodes*, it is unclear to what extent this is implemented in Intermind. Limited filtering is available: inside a channel the user can select topics to receive from predefined per-channel topics. Additionally, channels can be categorized according to user-defined categories. Our comments for Webcasting regarding the transport system and the scalability apply for Intermind too. Receiver software cannot be updated automatically. Distributed data is neither authenticated nor secured. Intermind owns a patent on push-like communication [10].

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. Before elaborating on this topic we will first contrast electronic mail and Usenet news with push systems and event-based systems. A combination of electronic mail and Usenet news can be used to emulate some of the functionalities but fails to meet the requirements of more advanced systems.

Mail has a 1:1 relationship model between producers and consumers. Even if mailing lists are used, 1 mail to *n* receivers is duplicated and transmitted *n* times. It is strictly decoupled and has very limited interaction functionalities. Usenet news has an n:m relationship model. The biggest problem of Usenet news is that it is no longer scalable without substantial redesigns [7]. Both electronic mail and Usenet news lack integrated concepts for authentication, secure transmission, mobile code, administration, etc. This list of deficiencies is not comprehensive but motivates the need for new concepts beyond electronic mail and Usenet news.

5.1 Mobile Code

Program code used to be bound to a certain processor / computer. The intention of the mobile code paradigm is to have code travel around networks and computers. So-called mobile agents 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 (pushlets) that is to be executed at the receiver. Thus push systems must address similar issues as pure mobile code systems although at a simpler scale since some of the problems in mobile code systems do not arise for push systems (routing of 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 a 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. 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.

5.2 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 [4]. 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 [23] that suggests a seven-dimensional design space. Some classifications of event-based approaches are given in [4].

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 4 lists the main differences between the two paradigms.

Table 4. Push systems vs. event-based systems

	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

The purpose of push systems is the timely distribution of data to consumers whereas 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 everyone 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, only have a very loose coupling between producers and consumers and therefore must have powerful mechanisms for event selection.

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 producer with little change. Though consumers are notified on new channels, for example via a meta-channel, subscription and unsubscription will occur infrequently once a satisfying profile of interest 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. 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 on occurrence of a pattern. Though the usefulness of this mechanism is undoubtedly high it adds consider-

able complexity to the implementation of such services. Ordering 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.

Our comparison shows that, despite their similarities, the differences in foci, requirements, and applied concepts, event-based and push systems are distinctly different models of distributed information systems.

6 Requirements for Widespread Use

We have presented the push paradigm 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, among these are: system-level design and integration issues, business-oriented issues, and multidisciplinary issues that reach beyond computer science and software engineering. In this section we present the main issues that we believe must be addressed in order for push systems to become usable on a wide scale. We also believe that these issues are relevant to any architectural model to be used on an Internet scale.

The key underlying design goal for any distributed system is *scalability*. The component model of Sect. 3 supports scalability by clearly separating producers from receivers by an intermediate transport system. We have shown a standard structure for the transport system based on caching and replication. As we have seen in Sect. 4, the model can be used to accurately describe the structure of existing push systems and 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 Sect. 3 can be used as a basis for developing a scalability model and reference implementations for push systems. We are developing such a reference implementation in the Minstrel project at the Technical University of Vienna. We are 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 10000 receivers over a typical mixed-bandwidth network is around 45 seconds while with standard non-optimized email this would take over 1 hour. The average delay would be around 13 seconds. 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 among the transport infrastructure). However, this provides a good indication towards performance figures because currently bandwidth is 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 there is control (e.g. on private networks).

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 sys-

tems, 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 (micro-payments, pay-per-view). In Minstrel, we are developing standard components for payment schemes. Figure 5 gives a model for a pay-per-view interaction in Minstrel.

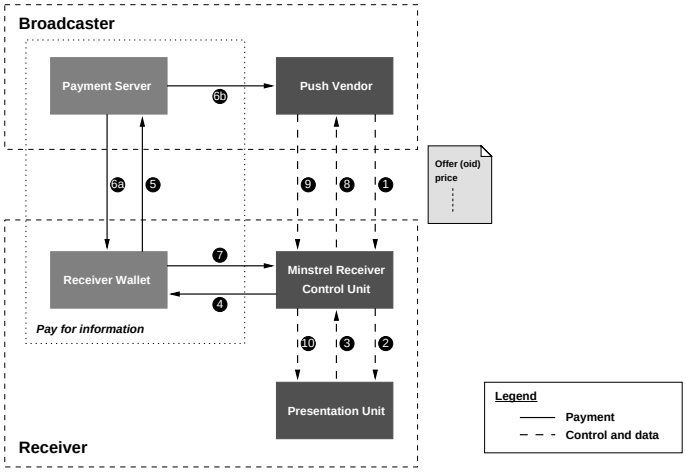

Fig. 5. Pay-per-view in Minstrel

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). Then 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.

Another business-related issue is *security and authentication*. If high-quality information providers 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 information from

an identifiable source. This issue is important in typical push application domains like news agencies, financial information services, and other high-confidence businesses. Technically this requires the availability of authentication frameworks, and certificate authorities and on a large scale (X.500, LDAP). For confidentiality of the data itself, encryption methods must be supported by the push systems. Full integration into push systems and "chain of trust" infrastructures—for example, you have to trust all sites running repeaters—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. As event-based systems, push systems can also facilitate software release management [25], and software deployment and configuration management [8, 9]. 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 [11].

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.

7 Summary and Conclusion

Even though there are many documents on the worldwide 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 paper 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. We have presented a component model for push systems that may be used to study, analyze, and contrast different implementations of push systems, and we have done that for six prominent push systems. Using the concepts of information source, receiver, broadcaster, and transport system, our component model separates the issues of content management, channel management, scalability, and user-interface management into different components. Our component model may be used as a basis for a reference implementation of push systems.

We have contrasted push systems with the closely related paradigm of event-based systems, pointed out the distinguishing features, and shown the connection with mobile code systems. We have also presented the main issues that need to be tackled by push systems: scalability, network traffic, security, authentication, and electronic commerce. We are currently addressing all these issues in our Minstrel project [19]. The Minstrel project uses the component model of Sect. 3 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 proof-of-concept implementation of our architectural model and serves as an extensible software platform for further research. The main design issues of Minstrel are scalability, a hybrid broadcasting paradigm that supports timely notification while requiring no special multicast infrastructure, a distributed model for simplifying information authentication, integrated support for micropayment e-commerce (e.g., Millicent), and support for pushlets that are executed in a highly configurable client security framework (subtractive security policies, security negotiation). Minstrel is being implemented in Java and the protocols are based on RMI.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable inputs for this paper.

References

[1] BackWeb. *BackWeb Polite Server*. BackWeb Technologies, 2077 Gateway Place, Suite 500, San Jose, CA95110, 1998. http://support.backweb.com/public/Version5.0/SERVER/INDEX.HTM.

[2] BackWeb. BackWeb – a cooperative architecture for a flexible push-pull broadcasting solution. Technical report. BackWeb Technologies, 2077 Gateway Place, Suite 500, San Jose, CA95110, March 1997. http://www.backweb.com/pd/whitepaper.html.

[3] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems – concepts and design*, International computer science series. Addison-Wesley, 1994.

[4] 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.

[5] 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.

[6] C. Ellerman. Channel Definition Format (CDF). Technical report. W3C, 3 March 1997. http://www.w3.org/TR/NOTE-CDFsubmit.html.

[7] 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/.

[8] 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). 269 278, May 1997. ftp://ftp.cs.colorado.edu/users/andre/papers/ICDCS97.ps.

[9] 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.

[10] Intermind. About Interminds Communications Patents. Intermind Corporation, 1999. http://www.intermind.com/materials/patent_desc.html.

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

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

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

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

[15] 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.

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

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

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

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

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

[21] PointCast. *Product documentation*. PointCast, 501 Macara Ave., Sunnyvale, CA 94086, 1999. http://www.pointcast.com/products/intranet/techresources/documentation.html?ibttechp.

[22] PointCast. *Technical papers*. PointCast, 501 Macara Ave., Sunnyvale, CA 94086, 1999. http://www.pointcast.com/products/intranet/techresources/techp.html?ibtdoc.

[23] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. *6th European Software Engineering Conference (ESEC/FSE '97)* (Zurich, Switzerland, September 1997). Published as Mehdi Jazayeri and Helmut Schauer, editors, *Lecture Notes in Computer Science*, pages 344–60. Springer, 1997.

[24] 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.

[25] Andre van der Hoek, Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Software Release Management. *6th European Software Engineering Conference held jointly with 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Zurich, Switzerland, September 22-25, 1997), pages 159–75, Mehdi Jazayeri and Helmut Schauer, editors. Springer Verlag, Berlin, September 1997.

[26] A. van Hoff, H. Partovi, and T. Thai. The Open Software Description Format (OSD). Technical report. W3C, 13 August 1997. http://www.w3.org/TR/NOTE-OSD.html.

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

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