A Programmable Structure for Pervasive Computing

Ingar Mæhlum Arntzen
Dept. Computer Science
University of Tromsø, Norway
ingarma@cs.uit.no

Dag Johansen
Dept. Computer Science
University of Tromsø, Norway
dag@cs.uit.no

Abstract

This paper presents the idea of a proactive and extensible pervasive computing infrastructure. It supports its users by filtering and pushing highly personalized context-aware information. We have derived a general structure that describes a virtual computer extensible at run-time by user-defined information filters. Individual users may program the structure using an expressive event-based filter programming model. In addition, the structure implements general design principles of push-based communication, by supporting both server-level and application-level extensibility.

1 Introduction

Pervasive computing is based on an integrated environment saturated seamlessly with computers, sensors and communication facilities. A typical user is moving about with his handheld or wearable computing device interacting with this environment. In this scenario, services related to communication and information access become essential. As the user walks about, he wants the pervasive infrastructure to *push* his communications (e.g. emails, phone-calls and news feeds) to a suitable device. For this reason, we suggest a proactive pervasive computing infrastructure that *pushes* highly personalized and context-aware information to its mobile users.

To accommodate this proactive and highly personalized environment, our infrastructure is *extensible* by individual users. This means that a single user may up-load and deploy personal software (instructions) on extensible servers. An information *filter* is the unit of extensibility, a filter being a small executable program. Information filters explicitly express the information interests of a user, i.e.*what* information a user wants and *when* he wants it delivered.

This proactive and highly personalized pervasive computing infrastructure requires solutions to a wide range of problems. The problem we address in this paper, is how to construct the basic building block of this infrastructure: A server extensible by individual users. We aim to derive a general structure for this type of servers.

The rest of this paper is organized as follows. Section 2 presents the WAIF[1] project motivating this research. The general structure we have derived for the pervasive infrastructure is presented in section 3. The rationale for our structure is discussed in section 4. Related work is presented in section 5, and the paper is concluded in section 6.

2 Wide Area Information Filtering

WAIF (Wide Area Information Filtering [12]) builds on over 10 years of experience with experimental mobile agent systems [10, 11]. Our TACOMA system was the first mobile agent system ever supporting agents written in almost any language. An important lesson we learned, is that this technology is very suitable for installing software components at run-time by single-hop mobile agents. Hence, we can structure servers with a static part exporting a service API to extensions, and a dynamic part consisting of user extensions using this API (mobile agents used as dynamic Web services).

This concept was used to move computations close to data sources like, for instance, moving a filter agent implemented in C and with a size of a few Kbytes, to a satellite data store containing Terabytes of data. Another application was a distributed sensor network, where we extended servers [8, 9] with small alert agents [7]. These filters parsed a continuous stream of weather and environment data close to the loggers in the Arctic, and alerted remote users based on highly personalized user predicates. The radioactivity level close to the Russian border was one important parameter to monitor for this StormCast system, extreme Arctic low pressures moving in from the Arctic sea another.

In WAIF, we focus on how the next generation pervasive Internet can be made programmable and extensible

[1] WAIF is a joint project between University of Tromsø, Cornell University, and UC San Diego. `http://waif.cs.uit.no/`
Funded by Norwegian Research Council (IKT-2010 Program).

0-7803-8577-2/04/$20.00 ©2004 IEEE

with personalized, mobile software. We conjecture that the Web's next paradigm shift will include a much more proactive computing model. This will transform a passive Web being searched by users, to information and service providers searching actively for users. Our goal is to replace the old, time-consuming pull-based Internet, with a pervasive, push-based one delivering high-precision, context sensitive information in a timely manner. Hence, we attempt to structure future generation distributed systems so that we get computers (and spam) as much as possible out of the (visible) human loop.

The infrastructure we build for this proactive Web service environment is extremely asymmetric. On the traditional server side is an integrated environment with computers, sensors and communication facilities. A typical client is a user moving about with his handheld or wearable computing devices interacting with this environment. This implies that the user environments should be moved along the trajectory of the user. The first WAIF prototypes support task mobility, user environments (i.e. desktops) transparently moving along the trajectory of the user. If required software is missing at a server, a server can, for instance, up-load and rent this from trusted third-party software vendors.

It is important to notice that a typical WAIF user is not a computer expert. As such, a user is configuring his environment with extensions written by somebody else. We use a multi-tier architecture approach, with the end user being a computer novice. Simultaneously, he is capable of installing software components in the pervasive infrastructure by explicit and implicit actions. These actions, including a user entering a room or logging in to a computer, map down to directories of software components that can be used. In this paper, we focus on aspects related to writing these components and how to provide an infrastructure for running them.

3 Structure

This section presents the general structure derived for computers in a proactive pervasive computing infrastructure. The structure describes a virtual computer, extensible at run-time by user-defined information filters. Such filters may (occasionally) push relevant information back to their users.

3.1 Application-level Extensibility

A template for an extensible server, based on our experience with TACOMA, is illustrated in Figure 1. The template has two main parts; *SERVER* and *APPLICATION*. The structure supports application level extensibility, by allowing individual users to install personal information filters (*F*). The server-part of the structure hosts the execution of these filters. In addition, it exports an API to running filters, including three generic services. First, filters may receive messages published by the server, at the *IN* buffer. Second, the *STORE* API allows filters to read or write persistent data. Third, filters may push information externally, using the *PUSH* primitive.

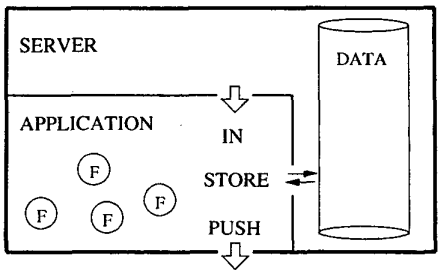

Figure 1. A template for the general structure.

3.2 Filter Programming Model

Filters describe the information interests of end users. An important goal of our structure is hence that filters are easily specified. At the same time, users also need unlimited expressiveness to specify their interests accurately. We advocate a solution, where filter programmers express interests using traditional programming language constructs. A high level programming language may be both easy-to-use and expressive. As a consequence, we say that users *program* their interests, i.e. their information filters.

3.2.1 Programming Filters

In our structure, a filter is defined to be an expression of user-defined, finite[2] functionality. Figure 2 illustrates a simple filter example, implemented by a Python function. The filter executes on incoming messages (published by the server), and implements user interests by pushing only messages of a certain type.

```
def filter (IN): # IN message
    if IN['type'] == 'sport':
        PUSH(IN) # PUSH to me
```

Figure 2. A simple filter example.

The body of a filter contains user-defined logic, expressed in a high level programming language. Hence, a

[2]Bounded execution time.

filter may read and write variables in memory, and use basic primitives for selection and iteration. It may even use advanced programming abstractions like e.g. object-oriented techniques to express its logic. In addition, many of the libraries provided by the programming language may also be accessible for the filter programmer.

Three types of services are made accessible for running filters; incoming messages, persistent storage and a push-primitive. An incoming message (published by the server) is passed as a parameter by the server, to a filter. As the filter executes, it may reference the message like a function references its parameter. Messages may, for instance, be stored, altered or pushed (to the user).

Filters are provided access to persistent storage by the *STORE* API. Figure 3 gives a simple example of this. Persistent storage is useful for a number of reasons. A filter may crawl or analyze it, and push interesting results back to its user. Also, if filters have write access to storage, they can store their own history and make later decisions based on that. Shared storage may be used to communication between filters. The server decides what storage resources are accessible for filters. For instance, the server can grant privileges on a per-user basis.

The overall idea of the *PUSH* mechanism, is to allow filters to push information directly to their users. On a lower level of abstraction, this implies that messages should be pushed to a computer in the user proximity. Alternatively, intermediate filter servers may be available for further filtering and forwarding of the pushed information. As illustrated in Figure 2, no destination URL is indicated for pushed information. This is because the destination is always the owner of the filter, be it an intermediate filter server, or the handheld device of a user. The identity of the filter owner is recorded during installation.

3.2.2 Programming Events

Users want to express *what* information they want, and *when* they want it. We approach this by suggesting a filter execution model where user-defined events trigger the execution of user-defined filters. Users express *what* information they want with a *filter programming language*. In addition they express *when* they want this information, using an *event programming language*. The general structure implements a simple event programming language, where user-defined events may be constructed from three generic server events. They are time events (T), incoming message events (IN), and condition events (C).

Time events bind the execution of a filter to a specific moment in time. Time events are expressions of time, (e.g. t==12.00) and they are published by the server when this expression is true. Time event may also be periodic, thereby allowing repeated execution of a filter. [3]

IN events bind the execution of a filter to the arrival of an incoming message. IN events are expressed by naming a message channel. The filter in Figure 2 is bound to an IN event.

Binding a filter to a condition event causes the filter to be executed when a given condition becomes true. The condition may be an expression based on persistent state. The server implements an event service that supports the definition and evaluation of such conditions.

```
# Delay all news, except during lunch.
def filter0 (news_item):
    t = Time()
    start_lunch = Time('1200')
    end_lunch = Time('1230')
    if start_lunch<t and t<end_lunch:
        PUSH(news_item) # to me
    else:
     STORE.add(news_item,'/delay')

# Push delayed news at noon.
def filter1 (time):
    news_items = STORE.read('/delay')
    STORE.delete_all('/delay')
    for news_item in news_items:
        PUSH(news_item) # to me

# Bind filters to events
filter0.type = 'IN'
filter0.channel = 'newschannel'
filter1.type = 'T'
filter1.time = Time('1200')
```

Figure 3. Two filters that collectively form a delay application.

The filter programming example in Figure 3 illustrates the use of these event types, and also how multiple filters may collaboratively form filter applications. If a user subscribes to a news channel, but wants to receive news updates only during his lunch break, this delay application will solve his problem. Two naive filter implementations are given, to illustrate the basic structure. One filter is bound to an IN event, and the other is bound to a T event. Together they form a filter application that delays news items until noon. The IN-filter executes every time a message arrives on the news channel. If the message arrives before lunch, or after, the message is stored temporarily. If not, it is pushed to the

[3] Functionality that runs continuously is problematic since filters are required to be finite. Such functionality may instead be represented as a finite filter repeated an infinite number of times.

user. The T-filter is responsible for pushing delayed news items as the lunch break commences. It does so by binding to a time event. When it executes, it pushes all delayed messages and resets the delay folder.
The filters are implemented in Python. A Python function is used as a container for the filter logic. Two attributes are assigned to these filter functions. The *type* attribute states the filter type, in this case IN or T. In addition, a naive event programming language allow us to define the triggering events by attaching an event expression. The IN-filter is triggered only if the incoming message arrives on a specific channel, and the T-filter is triggered only at a specific moment in time.

3.2.3 Filter Execution Model

A highly extensible filter programming model requires filters to be installed and uninstalled easily at run-time. A suitable filter execution model is essential for achieving this goal. Inspired by the event-based programming model discussed above, we suggest a filter execution model where all related filters are executed in a single virtual thread. The single thread subscribes to the event sequence provided by the server. Whenever an event occurs, the thread loads and executes all filters bound to that event, passing the event itself as input. The execution of filters is hence virtually sequenced.[4] Installing and uninstalling filters using this execution model become trivial. Installing a filter involves two simple steps. First, the filter logic is added to the collection of filters already present. Then, a mapping is created in the server between the event and the filter. This mapping is used by the server to execute the filter when the event occurs. Uninstall is the inverse action, including removal of the event-filter mapping and the filter logic.

To summarize, the filter execution environment of our structure allows a collection of filters triggered by T, IN and C events to form user-defined filter applications. In addition, this filter application is extensible at run-time by adding or removing filters from the collection. This improvement to the structure is illustrated in Figure 4.

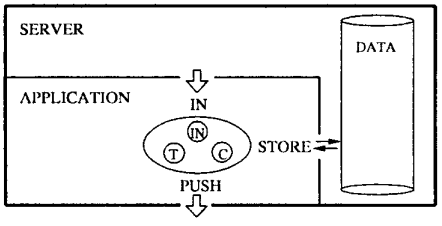

Figure 4. T, IN and C filters added to structure.

[4]This is one reason why filters are required to be finite.

3.3 Server-level Extensibility

The server-part is hosting the execution of application-part filters, providing them with run-time resources. It can also be a content provider that publishes valuable information to application-part filters. The server may do so by pushing messages to the IN buffer of application-part filters. Alternatively, messages may be stored persistently and made publicly accessible. In addition to publishing and storing, server functionality may need to receive messages from external sources. These messages may, for instance, be republished to application-part filters. For these reasons, the server-part functionality needs an IN buffer, a *STORE* API and a *PUSH* primitive, just like application-part functionality. Also, the server administrator may need to extend server functionality, so that new services can be added to the server.

This observation suggests that the filter programming model defined above is applicable also for the server-part of the structure. This idea is illustrated in Figure 5. The *PUSH* primitive is used by the server to publish information to application-part filters. An important implication of this is that publish-subscribe matching now happens internally in the structure.

The completed general structure describes a virtual computer extensible at run-time with both server level and application level filters.

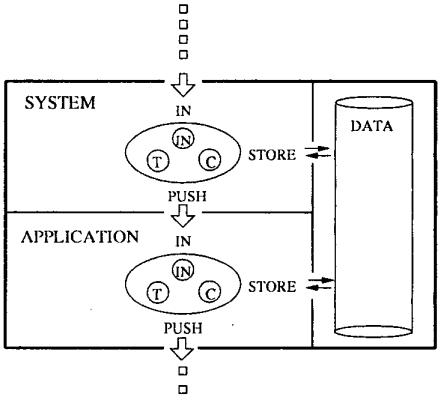

Figure 5. The completed general structure.

3.4 Implementation

We have implemented a prototype filter-server based on the general structure. The filter-server is implemented in Python. The structure allows filters, like those defined in Figure 3, to be installed and executed.

A Python function works as a container for filter logic. The server executes a filter by invoking this function with

the triggering event as parameter. The name of the function is not important for filter execution, but is used under installation to store the filter persistently. The name is required to uninstall the filter at a later time.

The implementation currently supports filters to be triggered by T events and IN events. T events allow filter execution to be bound to a specific moment in time. Currently, the event programming language is simplistic. The moment in time is given as a string, like e.g. *'08.00'*. Periodic T events are supported by providing an interval length in seconds. Filters triggered by IN events are executed as messages arrives. An IN event is defined by naming a message channel as a string. A message channel is essentially a description of what message type the filter expects. Filter functions are bound to a triggering event simply by assigning the event to a function attribute.

A filter is installed by enclosing it in an install message, and pushing that message to the IN buffer of the filterserver. Server functionality will then store the filter persistently, and create a server mapping between the triggering event and the stored filter. This mapping is used by the server to decide which filters should be executed, given the occurrence of a server event.

The overall functionality of the server is to monitor server events and map them to filter execution. A single thread implements this in an infinite loop. Periodically, it checks for incoming messages and matches T events with the server time. If messages have arrived at the IN buffer, a mapping (updated under installation) between IN events and filters is used to schedule the appropriate filters for execution. A message copy is given to each filter as a parameter. Similarly, if a T event matches the server time, the mapping between T events and filters is used to load and execute these filters sequentially. If a filter is periodic, the new execution time is computed by the server, and a new T event is created and bound to the filter. References to a simple storage API and a push mechanism are imported into the address space of executing filters.

This early prototype has a few notable limitations. Although programming and installation of simple filters is easy, difficulties arise when more complex filter applications are considered. For instance, if the user wants to build a complex pipe of filters (each filter is triggered by the output of the previous filter) this is possible in our implementation, but not easy. The next version of the filter server under development targets better abstractions for building complex filter networks. In addition, a more expressive event programming language is being developed, C events must be supported, the storage API much be richer, and better security must be provided.

3.5 Applicability

This subsection discusses two example applications using the general structure. The two applications represent an information source and an information sink. For this reason, they are typical examples of relevant applications in a pervasive computing environment. The information source is an *online newspaper*, and the sink is what we call a *personal filesystem*.

Figure 6 illustrates the following scenario. A journalist publishes news items by updating web-pages locally at the server of an online newspaper. The server then pushes these news items to readers that find them relevant. At the other end, the personal filesystem shields the reader from spam and information overload by interrupting and filtering of incoming pushes. Both the online newspaper and the personal filesystem are modelled using our general structure for push-based servers.

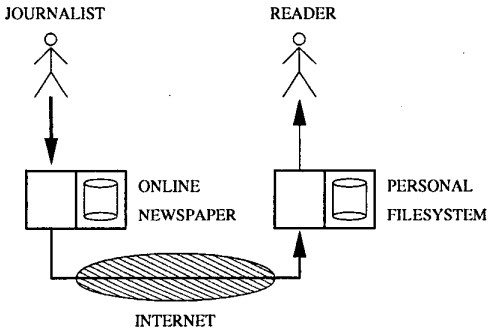

Figure 6. An Online newspaper pushes news items to a reader.

3.5.1 Online Newspaper

Online newspapers are a simple example of a back-end service in a pervasive computing infrastructure. News articles are published continuously 24/7, and readers typically navigate or search the content through a public Web interface. Although online newspapers provide valuable services on the Web, the limitations of the pull-based interaction scheme is evident. Readers waiting for news must poll for updates, and the newspaper can only serve its customer during that short second between the request and the reply. This motivates the construction of a push-based online newspaper. Readers want to specify accurately *what* news items they want, and *when* these news items should be pushed. This suggests applicability of our general structure.

Adopting the general structure leads to the following description of an online newspaper. The server-level function-

ality publishes news items on different channels (e.g. sport, politics). Readers may then extend application-level functionality with highly personalized information filters. For instance, a given reader may find it amusing to push all news items containing his first name. In addition, users may delay the push of news items until an appropriate time, (e.g. lunch break). A helpful newspaper documents its services, and possibly also provides filter templates.

The pull-based services of the online newspaper may also be extended and personalized. This is possible if server functionality republishes incoming requests to application-part filters. A single user may then extend the functionality invoked by the request. This way, personalization of pull-based services is also attainable.

3.5.2 Personal Filesystem

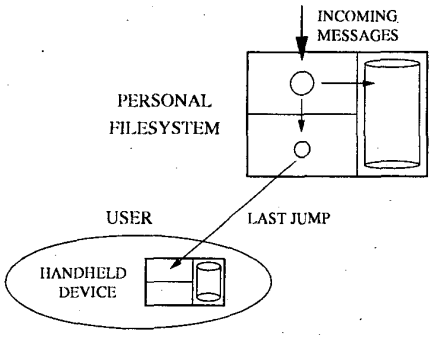

Figure 7. Last jump, from personal filesystem to handheld device.

In a proactive pervasive computing infrastructure, back-end services push information to thin computing devices in the user proximity. As a consequence, the classical problem of information overload is worsened for the user. Even if he does not receive any spam, he still has to decide what to do with the received information. It is crucial that the overhead with pull-based interaction is not simply transformed to information overload. The personal filesystem solves this problem by automating the receipt and storage of pushed messages. Only the most important messages should be pushed on to the user. This implies that the user must specify accurately *what* information is important, *when* it is important, and *how* the personal filesystem should deal with messages of lesser importance. Again, this suggests applicability of our general structure.

Figure 7 illustrates the following description of a personal filesystem. The server-part functionality filters incoming messages. This may include spam-filtering or automated archiving. For instance, a user may want to store incoming messages, in specific folders, in order to maintain the organization of his *home* directory. Important information should immediately be presented for the user. In the general case, the user is operating a mobile computing device. Pushing information to this device is possible if the device has installed a filter on the personal filesystem. If this is the case, the server-part of the personal filesystem simply republishes messages to application-part filters.

4 Discussion

This section discusses the rationale behind the design of the general structure described in section 3.

4.1 Filter Programming Model

If end users are to program the pervasive computing infrastructure, then the programming model must allow them to express their interests in a way natural to them. Our event-based approach is inspired by how people typically express their real-life desires. For example, the following three instructions might be relevant for a given user to install in his pervasive environment.

- "When I wake up, push me the latest headlines from New York Times".
- "When i receive a call, then mute my mp3 player".
- "When the number of unread emails from Bob exceeds 5, push an alert".

These examples illustrate that people naturally bind real-life events (*when*) to real-life activity (*what*), as they describe desires. Motivated by this observation, our filter programming model allows users to program the infrastructure by binding user-defined events to user-defined filters. Hence, users may express *what* information they want and *when* they want it, by programming personal events and information filters.

4.1.1 Expressiveness

A high level of expressiveness is required for the programming of a highly personalized filter environment. We approach this by suggesting that both events and filters are programmable using traditional programming language constructs.

Defining a filter programming language is not that difficult. A minimalistic language need only be able to manipulate data, use a *STORE* API, and a *PUSH* mechanism. Most languages are, but high level languages are preferable because they are easier to use.

The definition of an event programming language is more challenging. Users may, for instance, need to describe

real-life events (e.g. *"When I wake up"*) to fully express their desires. Our approach is to define a set of generic event types, T, IN, and C events. Users may program personalized events by specializing or combining these basic event types.

Programming with these event types need not be difficult. For example, the three real-life events from the above example may easily be expressed by T, IN or C events. The *"When I wake up"* event may for instance be approximated by a 7.am T event. If we assume that communication by phone is an integral part of the pervasive infrastructure, then the *"When I receive a call"* event is simply an IN event bound to the appropriate message channel. The third event, *"When the number of unread emails from Bob exceeds 5"*, may be expressed as a C event, where the evaluated condition is an expression over the emails in the inbox.

The expressiveness of the event programming language depends not only on the three basic event types. Especially, sensor inputs from the user environment may be very useful, when it comes to defining real-life events. For instance, the implementation of the *"When I wake up"* event might be improved if sensors from the bedroom (e.g. light switch or bedroom door) are taken into account. If sensors push their signals as messages to the IN buffer, sensor input is available using IN events.

The simplicity of this event-based programming model does not come without a cost. For instance, users may unknowingly install filters that conflict with existing functionality. A consequence of the simple programming model, is that the structure must be responsible for correct sequencing, prioritization and conflict resolution. This may be difficult, hence an optimistic solution might be more appropriate. If the structure can detect conflicts and ambiguities as filters are installed, the responsibility for solving them can be given to the user.

4.1.2 Extensibility

Extensibility and adaptability of user-defined functionality at run-time, is important for several reasons. For example, user mobility implies that functionality must be relocated or reconfigured dynamically. In addition, the user may change his information interests or context frequently.

Our event-based programming model is extensible in the sense that filters may be installed or uninstalled at run-time. Still, how filter applications should be programmed, in order to be extensible in desirable way, is another problem. The trivial solution to extensibility is for the user to implement all his functionality in one huge filter (possibly triggered by a multitude of events). Changing some part of the functionality then means uninstalling, reimplementing and reinstalling the entire filter. This might be less desirable, especially if updates are small and frequent. Instead, the idea is to split the logic of the huge filter into smaller filters and update them independently. This way, a large filter application can be built from many single-event filters. The problem with this solution, is that splitting logic often introduces dependencies. For instance, the execution of one filter may rely on some variable produced or updated by another filter. Solving this problem implies implementing communication between filters.

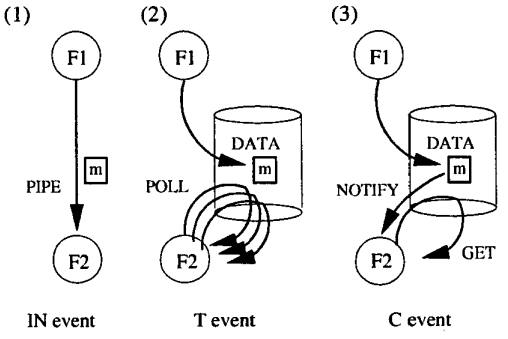

Figure 8. Three alternatives for filter communication.

Three different ways of communicating a message *m* between two filters (F1) and (F2) (executing at different points in time), are illustrated by Figure 8. *(1)* If real-time performance is required, F1 may *PIPE* the message through memory (thereby forcing the execution of filter F2 directly after F1). Alternatively, *(2)* and *(3)*, the message may be communicated via shared persistent storage, i.e. filter F1 writes the message to *STORE*. Filter F2 may then access this message in two ways. Either, *(2)*, F2 may *POLL* for the arrival of *m*, or, *(3)*, the system may *NOTIFY* F2 when *m* has arrived. In the first case, F2 invokes the *GET* method of the *STORE* API repeatedly (until *m* arrives and *GET* returns successfully). In the second case, *GET* need only be invoked once.

An interesting observation is that the three ways to do filter communication correspond directly to the three generic event types defined by the structure. To *PIPE* a message to F2, implies that F2 must be triggered by an IN event. If F2 is triggered by a periodic T event, it may *POLL* for the arrival of a message, and if F2 is triggered by a C event (condition: *m has arrived*), F2 may *GET m*. This represents a more formal interpretation of the three generic event types.

4.2 General Push Communication

The general structure is designed for a push-based pervasive infrastructure. Based on a general discussion of push-based communication, three general design principles are proposed for push-based applications. The derivation of the general structure has been motivated by these principles.

This discussion of push-based communication is centered around a simple model illustrated by Figure 9. Two servers, a *source* and a *sink* engage in a push-based conversation. The source publishes information, and at least some part of that information is relevant to the sink. The goal (possibly shared by both parties) is that the source pushes all the relevant information to the sink, and nothing but the relevant information. In order to reach these *recall* and *precision* goals, several problems must be addressed.

4.2.1 Principle 1

The first problem is how sinks may define their information interests. In order for the source to push relevant information to the sink, it must have prior knowledge of sink interests. The main idea of this paper defines our approach to this problem. We want the sink to *program* the source, using our filter programming model. This way, the sink gets the expressiveness needed to define its interests accurately. It also gets the extensibility needed to change interests frequently. In addition, the sink has a way of specifying *when* a push is appropriate. Translating this to the terminology of the general structure means that application level extensibility for individual users, is required for the source.

Design principle 1 *A source should be programmable (extensible).*

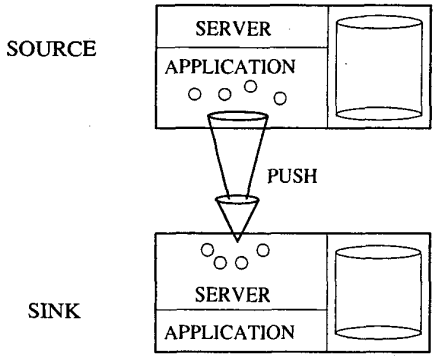

Figure 9. General model of push communication.

4.2.2 Principle 2

A second problem is that of protecting a sink from a *spamming* source, or from information overflow. First, given that a sink has successfully programmed its information interests with the source, there is no guarantee that these interests will be respected. The sink needs a receive-side filter mechanism that gives it absolute control over the incoming pushed information. This control mechanism must be simple, expressive and extensible. We approach this problem by suggesting that the sink also is programmable. In the terminology of the general structure, this means that server-level extensibility is required for the sink.

Another reason relates to the problem of information overflow. When the publishing of information is automated at the source, this introduces an imbalance if receiving pushed information must be handled manually. A consequence of this may be push-slavery, where the user is forced to react to every received push. Allowing the user to program the receipt of pushed information at his computing device, may ease this problem. For example, not all received information may require the immediate attention of the user. Only the most important information should be pushed on to, for instance, a screen or a loudspeaker.

Using our filter programming model makes it easy to define useful receive-filters. Imagine, for instance, a parameterizable filter that drops a certain percentage of the incoming messages. If this filter is updated using a *volume slide bar*, then the user may tune real-time the amount of pushes received at a given channel.

Design principle 2 *A sink should be programmable (extensible).*

4.2.3 Principle 3

The third design principle comes from the observation that source and sink roles may change frequently in a push-based application. For instance, a server may act both as sink and source simply by republishing some of the information that it receives. This applies for end-users as well. For this reason we suggest that a server is programmable both as a sink and a source, and that these filter environments are stacked so that received pushes may subsequently be republished to other users. In the terminology of the general structure, this means that both server-level and application-level extensibility are necessary for push-based servers.

Design principle 3 *A server should have both source and sink capabilities.*

4.3 Generality of Structure

This paper claims generality for the structure presented. The following arguments support that position. First, the structure is applicable for both extremes of push-based communication, the information source and the information sink. Second, the structure is applicable for all types of computers participating in a pervasive computing infrastructure, ranging from light handheld devices to clusters of

heavy back-bone servers. Only the implementation of the structure will differ. Third, the structure may host different types of push-based applications. Applications vary with respect to communication semantics such as guaranteed delivery (high recall) or heavy filtering (high precision). They also differ with respect to push-policies like *targeted* push or *solicited* push. We find that all these variations may be implemented in our structure by partitioning the filters, between the server part and the application part of the structure. Finally, pull-based communication may also be implemented using this structure. A request-reply protocol is achieved by pushing requests to the server-part, and receiving replies (as pushes) from the application-part. The functionality invoked by these requests may then easily be extended by individual users.

5 Related Work

WAIF leverages ideas, concepts, and techniques from several research areas. Most notably is work in pervasive computing based on the seminal work by Mark Weiser [16]. A large body of work in, for instance, ad hoc networking, wireless computing, handheld and wearable computing, and human-computer interaction are being done in this context [14]. WAIF fits into this large body of work by focussing on event-oriented operating system and Web server implementations, overlay distribution networks, mobile code, and non-functional aspects like, for instance, fault-tolerance and security for this type of computing.

Extensibility has been applied to computer systems for many years. Extensible operating systems and active networks contain a body of related work. For instance, SPIN [2], VINO [15], and Exo-kernel [6] demonstrate how to dynamically extend operating systems at run-time. A system like STP [13] uses un-trusted mobile code to upgrade communicating end-hosts at the transport level. A key difference with WAIF, though, is that we extend user space Web servers at run time, not in-kernel or protocol functionality. This is similar to how Web services in, for instance, Microsoft .NET can be used, where Internet services can be composed from Web services.

A push-based Web is not a new idea. For instance, publish-subscribe systems like Gryphon [1], Siena [3] or SCRIBE [4] push information to remote users. These systems provide topic-based, or limited content-based, subscriptions to published information, while our ambition is to push context-aware content based information from multiple sources. In addition, we *program* (extend) the remote filters in such a fusion network with user specific code.

The ambitious Oxygen [5] project at MIT also targets creation and development of a pervasive computational environment. WAIF shares many of the same problems, but our approach basically differs by our stronger focus on extensibility and structural issues of a push based Web.

Finally, recent programming languages provide mechanisms for extensibility at run-time. This includes, for instance, Java and its reflection mechanism. We do not develop a new programming language for this type of computing, but have the pragmatic approach that filters should be implemented in one of the existing programming languages best solving the concrete application problem.

6 Concluding Remarks

Extensibility is a fundamental principle in distributed computing. This paper presents an approach to server extensibility in a push-based and highly personalized pervasive computing infrastructure. We have derived a general structure for this kind of servers.

The structure supports application-level extensibility, by allowing individual users to install personal information filters. These filters implement the interest of users by pushing relevant and context-aware information. We propose an easy-to-use filter programming model, where users bind the execution of personal information filters to personal (real-life) events. Access to traditional programming language constructs gives users the expressiveness needed to explicitly and accurately define interests. Three generic event-types are suggested for the programming of personal events; *time-events*, *in-events* and *condition-events*.

The server-level extensibility of the structure, is motivated by general design principles of push-based communication. Problems related to spam and information overload limit the applicability of push-based technologies. We suggest a programmable approach to protection of servers (and users). Server-level extensibility allows a user (administrator) to install personalized receive-side filters. In addition, services provided for the application-level filter environment, need also be programmable. Our filter programming model is applicable in both cases.

The authors thank Fred B. Schneider, Robbert van Renesse, and Keith Marzullo for providing feedback to an early version of this paper. In addition, Åge A. Kvalnes, Nils P. Sudmann, Håvard D. Johansen, and students at the WAIF group have taken active part in forming the ideas presented in this paper.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-based Subscription System. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–61. ACM Press, May 1999.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating

System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–283. ACM Press, 1995.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a wide-area Event Notification Service. In *Transactions on Computer Systems*, volume 19, pages 332–383. ACM Press, 2001.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. In *Journal on Selected Areas in Communications (JSAC)*, volume 20, pages 100–110. IEEE Computer Society, Oct. 2002.

[5] M. L. Dertouzos. The Future of Computing. *Scientific American*, 281(2):36–39, Aug. 1999.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266. ACM Press, 1995.

[7] K. Jacobsen and D. Johansen. Ubiquitous devices united: enabling distributed computing through mobile code. In *Proceedings of the 1999 ACM symposium on Applied Computing*, pages 399–404. ACM Press, 1999.

[8] D. Johansen. A Distributed Approach to the Design of Applications. In *Proceedings of the 5th International Conference on Computing and Information*, pages 195–201, Sudbury, Ontario, Canada, may 1993. IEEE Computer Society.

[9] D. Johansen and G. Hartvigsen. Architectural Issues in the StormCast System. In *Theory and Practice in Distributed Systems*, pages 1–16. LNCS 938, Springer, 1994.

[10] D. Johansen, K. J. Lauvset, R. van Renesse, F. B. Schneider, N. P. Sudmann, and K. Jacobsen. A TACOMA Retrospective. *Software Practice and Experience*, pages 605–619, 2002.

[11] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HOTOS-V)*, pages 42–45, Orcas Island, WA, May 1995. IEEE Computer Society.

[12] D. Johansen, R. van Renesse, and F. B. Schneider. WAIF: Web of Asynchronous Information Filters. In *A.Schiper er al.(Eds.): Future Directions in Distributed Computing*, pages 81–86. LNCS 2584, Springer, 2003.

[13] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocols using untrusted mobile code. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 1–14. ACM Press, 2003.

[14] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, pages 10–17, Aug. 2001.

[15] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227. ACM Press, 1996.

[16] M. Weiser. The computer of the 21st century. *Scientific American*, pages 94–100, Sept. 1991.