View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Munin - Open Research Archive

A Programmable Structure for Pervasive Computing

Extended Abstract

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 exstended abstract presents an asymmetric and programmable (extensible) approach to pervasive computing. The idea is to off-load computations from light portable clients into a back-bone of seamlessly integrated servers. This way, a user can extend and personalize his pervasive computational environment by installing computations following his trajectory throughout the day. Focus on this extended abstract is on structural issues related to the back-end servers running mobile code off-loaded from the mobile user.**

I. 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. We embrace this asymmetric computational model, but focus strongly on making it proactive and highly personalized. The overall goal is to get the computers as much as possible out of the human loop, and this is best accommodated by a pervasive infrastructure *pushing* high-precision, context sensitive information to its users.

Additionally, we are interested in moving a user's computational infrastructure around. Once a user leaves a desktop or embedded device supporting his computational needs, his computations should be hoarded and brought along. This includes moving data like, for instance moving an mp3-based song around between audio players, and this without re-staring the song at each place visited. At the other extreme, we are moving commodity software like, for instance, Microsoft Powerpoint and Word documents between the nodes a user visits. The idea is to create a state-full, personal environment at the fingertips of the user in a pervasive environment. We have already demonstrated this concept using USB-memory sticks, and by using a more client-server based central server solution like in Aura [8].

To accommodate this proactive and extensible pervasive infrastructure, we suggest that a collection of *extensible servers* should form the back-bone. Hence, this back-bone network should be made programmable through extensions, computations off-loaded and installed from the handheld devices. This is not a new idea, since we have used our series of TACOMA mobile agent systems the last 10 years to off-load computations from light clients like, for instance, PDAs, cellular phones and portable computers [13]. What is new, however, is that we move along, install and cluster large collections of personal software and data around a portable user.

This type of software mobility has serious technical problems, be it a mobile agent or a filter being installed remotely. The problem we address in this exstended abstract is how to construct a web server that can be programmed (extended) with user code. Our experience from, for instance, extending filter and sensor network servers with single-hop mobile agents made us derive a general structure for this type of programmable servers. This fundamental structure can be viewed as a virtual computer running user extensions. This structure (and applications derived from it) is the subject of this extended abstract.

The rest of this extended abstract is organized as follows. First, we present the overall ideas in the WAIF[1] project. In section III our general structure for personal, programmable, push-based servers is presented, followed by two derived WAIF applications. We are currently building a new series of WAIF server prototypes and applications, and the full paper intends to add much more experience details.

II. WAIF

The WAIF project investigates structuring techniques for future generation distributed applications. Motivated by inadequacies of the current Web, the WAIF project is building a push-based Web [14].

A. Push-based Web

The idea of a push-based Web is motivated by limitation of the current Web architecture. The intention of the Web [3] is to allow Internet users world-wide publish and retrieve information easily. Although the Web is often believed to fulfill this intention, some serious limitations are becoming increasingly evident. Especially, the growing amount of published real-time information [14] represents a great challenge to the current solution. The reasons for this are problems associated with the pull-based interaction scheme, induced by the underlying client-server architecture. For instance, when a user (or an application) needs real-time discovery of events published by a remote source, they are forced to poll the source until the event occurs. This has several unpleasant

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

implications. First, from the user perspective, polling may be time-consuming if it is not immediately rewarding. Second, if users and applications all have to poll real-time information, this will seriously threaten the scalability of both the Web and the Internet. Third, since users are present on the Web only as they actively pull information, this makes it very hard for content providers to locate and service their customers. The consequence is that users must do all the work associated with locating and retrieving information.

One solution to these problems is designing a push-based Web, where users may subscribe to remote events, instead of polling them continuously. This improves scalability, and leaves the users with little overhead. The subscriptions form the links in the new Web, and when events occur at a publisher, information is pushed along these links. In contrast to the pull-based Web, this means that links capture relationships based on how information is used, rather than how the publisher intended it to be used [14]. A push-based Web may be crawled and indexed much like the current Web, but in the push-based Web, mechanisms for publishers to locate subscribers may be equally important as the opposite.

A push-based Web is not a new idea. For instance, publish-subscribe systems like Gryphon [2] and Siena [6] fit the above description. These systems provide topic-based, or limited content-based, subscriptions to published information. Global knowledge of publishers makes it possible to guaranty data delivery and perfect recall. This property is required by a wide range of applications, but in an information space like the Web, perfect recall makes no sense. In contrast, users of the Web often accept only the most relevant information. This calls for a push-based Web that also focuses on high precision. In WAIF we do that, by filtering information as it is pushed towards the user.

B. *Web of Asynchronous Information Filters*

The WAIF project is concerned with building a programmable and push-based infrastructure supporting a push-based Web.

In WAIF, we focus on the single user and aim to provide him with highly personalized and relevant information, in a timely and push-based manner. We do this by allowing individual users to offload personal information filters into a programmable and push-based infrastructure. Collectively these filters implement the information interests of a single user. They are executed asynchronously, and occasionally they push published real-time information back towards the user.

Providing relevant information is a great challenge, and it implies maximazing both recall and precision. A mechanism for locating new sources of information, and installing filters on them, is essential for providing high recall. Precision is achieved by pushing information through a fusion network of cooperating filters. In WAIF, a single mechanism is provided to help users locate information, install filters and connect them into a fusion network. We call this a PONS, a Personal Overlay Network System. The PONS produces a stream of highly relevant information surfacing at some device in the user proximity.

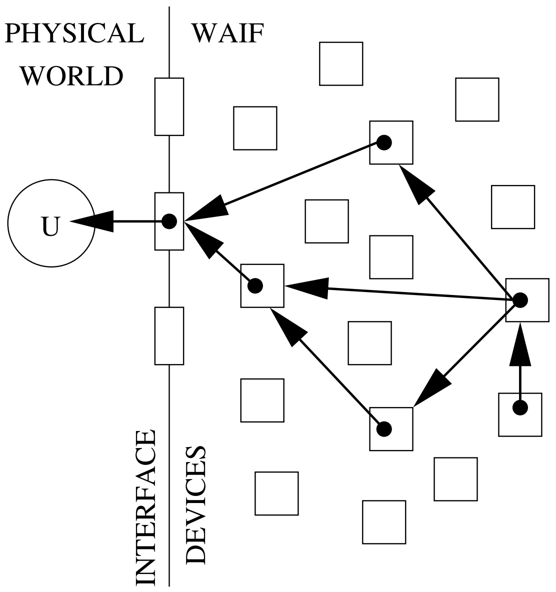

Fig. 1. The WAIF PONS.

Technologies for locating sources of information is an integral part of the PONS. In fact, all kinds of existing pull-based search technologies are applicable. The sources in the push-based Web may be browsed, indexed or sectioned by directory services. Still, push-based search technologies like recommender systems seem more appealing, from a research point of view.
Another challenge is to locate filters. For security reasons, it may not always be advisable that users actually write their own filters. Instead, users may locate code written by trusted 3'rd parties. This may also be appealing for users with little programming experience. For many WAIF publishers, providing appropriate filters may be anatural part of their service.
How basic filter patterns, *nanofilters* [14], may be derived, and later parameterized, stacked and combined according to user interests, is an important research issue in WAIF.

Allowing multiple users to install and interconnect filters in an unguided way, may have severe consequences for the system. Especially, the PONS must implement scalable sharing of information, this being the main motivation for the push-based Web. In WAIF we do this by stacking PONS filters in accordance to a stepwise refinement policy. Topic-based filters have general applicability, and may be shared by many users. For this reason, topic-based filters should run close to the source of information. Intermediate filter-servers may offer content-based filtering, to a limited amount of users. Context sensitive filters are required to perform the ultimate personalization. Since this functionality is less likely to be shared, it is reasonably placed close to the user.
Installation and interconnection of filters may well be hidden

for the user, or even automated.

WAIF is an application for a future pervasive computing environment. We share the vision of Mark Weiser [17], that users will interact with future computing systems using light devices, either woven seamlessly into the surroundings, or carried along. In such an asymmetric computing environment, the infrastructure must support migration of the computations of individual users. The WAIF infrastructure does exactly this, focusing on computations related to push-based information retrieval.

III. Structure

This section presents a general structure for a personally programmable push-based virtual computer. For simplicity, we collapse the first three words, *personally programmable push-based* into the acronym 3P. This means that a 3P computer allows individual users to extend its functionality by installing and running code. The code reflects personal computational interests, and may push results to its user. These properties makes 3P computers well suited as back-end computers in a pervasive computing infrastructure.

We have derived a general structure for 3P computers, a 3P structure. This is done by building real WAIF applications, and by leveraging experience from research in mobile agents [13], [15].

A. Motivation

The need for a general 3P structure is motivated by the following considerations. Experiences from mobile agents research show that these applications are hard to program, and that agreement on a complete operating and programming environment is required for wider acceptance and applicability [12]. This is even more so, when individual users are to program a multitude of filter servers, ranging from high end filter clusters to low end hand-held devices. The 3P structure provides a uniform programming model for 3P servers in a pervasive computing infrastructure.

B. Problem

The main problem of push-based information services is making appropriate push-decisions, at all times. *Relevance*, *expressiveness* and *adaptability* are important properties in this matter. Existing push-technologies are discussed briefly with regard to these properties.

An ideal information producer pushes all the *relevant* information to its consumer, and nothing but the *relevant* information. This corresponds to high recall and precision, respectively. To be able to do this, the information producer needs a description of consumer interests, as a basis for push-decisions. The *expressiveness* offered by the description language should allow the consumer to express interests precisely, yet easily.
Another important property is the *adaptability*. If the consumer suddenly changes his interests, it is critical that his interests are quickly and and easily updated. An interesting special case is when the consumer looses interest, completely or temporarily. It should then be possible for the consumer to close the push communication. In some cases *expressiveness* and *adaptability* are conflicting goals.

A number of attempts have been made to implement push-based information services, not all of them a great success. For instance, online newspapers have tried to push news items by email. Although intuitively a useful service, reducing the need to poll the newspaper website, it has not gained wide popularity. The *expressiveness* of such services is typically limited to a topic-based subscription scheme. Even worse is the low *adaptability*. Tuning of news interests is not provided through the receiving client, the email-reader. Instead, readers usually must update a user profile at the newspaper website. Filtering email on the email server is another approach, but here too, the user friendliness is relatively low. There is often a fine line separating relevant news from *spam*, and the combination of low *expressiveness* and low *adaptability* is not a good one.
Another approach to implementing push communication, is just offloading polling from the consumer to the consumer client. Publishing real-time updates on sport events using the web browser is one example. Another example is publishing of news items on RSS[2] feeds. Since both the polling and control mechanisms may be implemented by the client, the *adaptability* may be better for these solutions, compared to email.

A discussion of publish-subscribe systems, and recommender systems will be included in the final paper.

C. Idea

A major limitation of current push-based technologies is that consumers are not given appropriate tools to control the flow of pushed content. We advocate a solution where the consumer is given ownership and absolute control over the push communication. By controlling the *push-channel*, we assert that the consumer must control both what information goes into it, and what is filtered on the way. This corresponds to controlling recall and precision, respectively.

The WAIF idea is to maximize both the *adaptability* and the *expressiveness* of the *push-channel*, by allowing consumers to upload and execute code asynchronously at the producer side. This code makes both push and filtering decisions on behalf of the consumer. In fact, *only* consumer code has access to the *push-channel*. As a consequence, the producer is no longer concerned with push-decisions. Instead, he services the consumer by providing a rich executing environment for the consumer code.

This solution allows for a high level of *adaptability*. By differentiating between code that pushes information and code that filters information, both recall and precision may be tuned by the consumer. In effect, the consumer may implement a personal control policy. We call these code types *generating* code and *reducing* code. Separation of concerns is another reason for distinguishing *generating* and *reducing* code. Often,

[2]RSS: Rich Site Summary, Really Simple Syndication, RDF Site Summary.

push-decisions are a function of content alone, while filtering decisions may be based on context as well. The special case of closing the *push-channel* temporarily is easily solved by installing a filter that discards all pushes.

Consumer interests are expressed explicitly by writing both *generating* and *reducing* code. This way, the *expressiveness* of the solution is limited only by the programming language, and the limitations enforced by the execution environment. We believe that this level of *expressiveness* by far outperforms current push-based technologies. In addition the *expressiveness* enhances *adaptability*, instead of damaging it.

D. Requirements

Motivated by the above discussion, we list some general requirements to the 3P structure.

1) User code must be easily installed/uninstalled.
2) User code must run asynchronously.
3) The service must sequence *generating* and *reducing* code correctly.
4) User code must have access to a push mechanism.
5) A user must only receive pushes generated by code owned by himself.
6) The service may support multiple users.
7) User code may have private storage.
8) User code may have access to stored data.
9) User code may have access to incoming data streams.

E. Structure

An overview of the general 3P structure is illustrated by figure 2. The structure describes a logical computer, and does not specify its physical implementation. The logical computer has two main parts, *exec* is where user code is executed, and *store* is the persistent storage accessible for the user code. The *in* and *out* buffers are interfaces to the extern world. For instance, a message pushed from another producer arrives at the *in* buffer, where it is made accessible for user code. The *out* buffer is used when user code decides to push a message to its user, or another 3P server. A simple information filter is presented by figure 3, as a trivial example of user code. The filter is started with a reference to the incoming message, and makes a decision to push or discard it, on behalf of its owner. User code with more complex functionality may also access the storage.

The 3P structure describes a virtual computer, and does not specify its physical implementation. For instance, the 3P structure may be implemented by a large cluster, or opposite, a single CPU may host a stack or network of 3P structures. The overall functionality of the 3P virtual computer is filtering data, and consequently it may be seen as a large 3P filter. The structure may therefore be applied recursively, and on different abstraction layers.

1) Extensibility: The structure requirements 1 and 2 states that push-based services must be programmable by individual users. This means that the structure must implement a mechanism that allows users to install, run and uninstall user code easily, at any time.

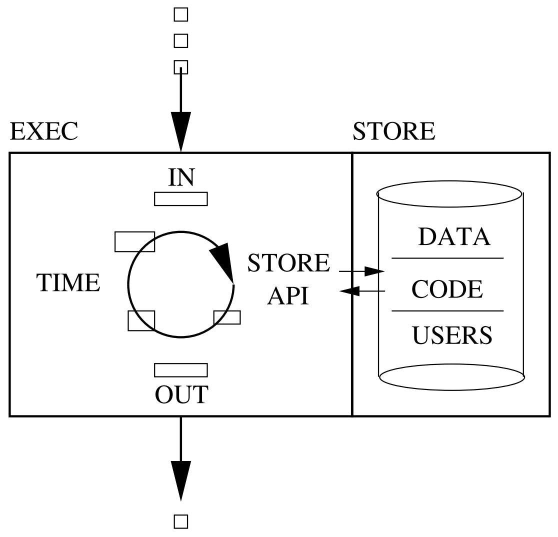

Fig. 2. The general Structure.

We achieve this level of extensibility by requiring user code to be expressed as a collection of finite tasks. In addition, the execution is triggered by system *events*. A single, infinite, virtual process may then sequentially execute tasks according to the order of the triggering events. For example, the code in figure 3 is a single finite task. The system generates a specific *event*, when an incoming message has arrived, and this *event* triggers the virtual process to load and execute all tasks bound to this event. This way, the resulting sequence of finite tasks may be viewed as a continuously running user application. In figure 2 this sequence is represented as a circular thread of control, with three tasks scheduled for execution.

Installing and uninstalling user code in this scenario is easy. When user code is installed, it is simply added to the collection of tasks owned by a user, and its execution is bound to the occurrence of a specific system *event*. Uninstalling user code conversely involves removing tasks from the collection, together with the corresponding event bindings.

```
item = self.in.get()
if item['topic'] == 'sport':
   push(item)  # to me
```

Fig. 3. A Python user code example.

2) Execution: The execution of tasks is triggered by system *events*. Four different *event* types are identified as essential to the structure: They are *TimeEvents*, *InEvents*, *OutEvents* and *DataEvents*.

- *TimeEvents* allow code execution to be scheduled at a specific moment in time. For instance, a user may program a filter to delay certain pushed items until after work. *TimeEvents* also allow code execution to be periodic. This makes *TimeEvents* an integral part of the execution model, since this is a way of implementing iteration. This is discussed further in section III-E.5.

- *InEvents* allow code execution to be triggered by the arrival of an incoming data item. This mechanism is essential for subscribing to, or filtering real-time data streams, and hence implements requirement 9. This mechanism embodies the very motivation for this work. A code example is given by figure 3.
- *OutEvents* imply that a data item is pushed to the *out* buffer by a *generating* task. By binding code to *OutEvents*, the code may override earlier push-decisions by discarding the buffered data item. This way a user may close a stream of pushed items temporarily, without uninstalling any code. Binding *reducing* code to the *OutEvents* ensures the sequencing demanded by requirement 3.
- *DataEvents* are associated with storage activity. For instance, code may be triggered by updates on specific data entries, performed by other tasks. This mechanism may be used for synchronization of task execution.

3) Push mechanism: Requirement 4 and 5 state that a push mechanism must be provided by the structure, and that it must ensure that user code may only push data items to its owning user. Note that this does not imply that multicast push is forbidden. It only means that *each* receiver of the multicast must install personal code subscribing to the multicast data.

In figure 3 a push primitive is provided by the programming language for invoking the push mechanism. The receiver of the push is always the owner of the code, so no receiving address is necessary. Each user has to register its identity on the system, previous to installing any code. This identity is referenced as owner by all installed code elements, and is used by the push mechanism. The structure hence implements requirement 6 if more than one identity is accepted by the system.

4) Storage: The storage has several roles to play in the structure. It is clearly necessary for stateful user code to store data persistently. This is expressed by requirement 7.
Equally important is viewing storage as a resource for the code. After all, the contents of the store may be the reason why code is installed in the first place. If requirement 8 is met, user code may be allowed to e.g. crawl, analyze or aggregate read-only data provided by the host.

In order for user code to access storage easily, the structure must provide a uniform storage API to all persistent data. A general graph storage abstraction is chosen because it is flexible enough to implement both the hierarchic filesystem and the relational database [9], [11]. In addition, the graph edges are a natural placeholders for meta-data, and allows for advanced search facilities to be implemented. The data model may be extensible, so that application-specific higher level data models may be constructed.

5) Programming model: Requiring code to be expressed as finite tasks with event bindings has implications on the programming model. Especially, an infinite loop within a task is illegal. Instead, the programmer must make use of the iteration services provided by the structure. A task may be iterated periodically by binding to *TimeEvents*. If the body of the loop is finite, this body should be selected as a task. This means that user code may have to be split into several tasks, with different event bindings, that collectively forms an application.

The mapping from events to tasks is essential to the overall control flow. For instance, a single *InEvent* may trigger multiple tasks. In some of these cases it may be necessary to specify a sequence for the execution of these tasks. Another problem is that these tasks may not be waiting for the same type of data items. In order to execute only relevant tasks, a guarding expression should be supplied. It is possible to organize these expressions in a hierarchy, for efficient event-task lookup.

6) Implementation: We are currently implementing a prototype 3P structure in Python. Our intention is to build the 3P general enough to demonstrate a wide range of WAIF applications, including the online newspaper and the *Personal Filesystem* discussed in section IV.

Non-functional requirements like efficiency, scalability or security will not be important features of the prototype, although we acknowledge that properties are essential for deployment. Instead, the final paper will bring a more detailed description of implementation aspects, the execution model and the programming model. We will also provide a richer set of user code examples to illustrate the flexibility of the structure.

IV. Applications

This section discusses two applications of the general 3P structure, that we are currently building. The two applications are important components in a pervasive computing infrastructure, an information producer and a consumer. The prototype producer application is an *Online Newspaper*, and the consumer is a *Personal Filesystem*. The online newspaper pushes news items directly to its reader, via the *Personal Filesystem*. Figure 4 illustrates this, and that both the online newspaper and the *Personal Filesystem* are modelled by the structure described in section III. The stream of news is filtered, both at the newspaper, and at the *Personal Filesystem*.

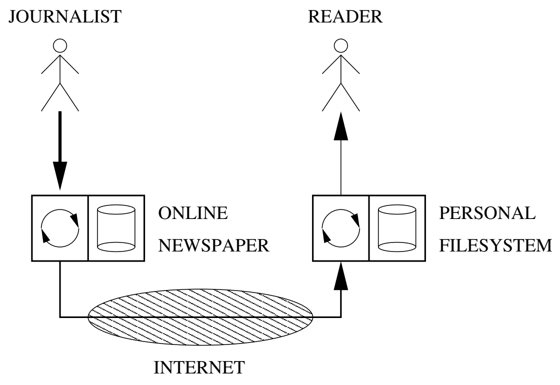

Fig. 4. Online newspaper pushing news items to a reader.

A. Online Newspaper

Online newspapers are a simple example of a back-end service in a pervasive computing infrastructure, that can push information to a portable client. Articles are published continuously 24/7, and readers may navigate or search the content through a Web interface. Still, for online newspapers the limitations of this pull-based interaction scheme are evident. Readers waiting for news are forced to polling, and the newspaper can only serve its customer during that short second between the request and the reply.

The motivation for building a push-based online newspaper is clear. Readers want to specify what and when news items should be pushed to them, they need to filter out spam, and they need to create their own personalized services. Future customers may even accept to pay for this, so there is a real incentive to build push-based online newspapers.

1) Push: Adopting the general structure leads to the following description of an online newspaper. All news items are published to the *in* buffer. The newspaper provide its own code to archive all published news items correctly. Each reader may install personal code on the system. This code has read access to the archive, and a limited private storage. Readers may bind code to the *InEvent* and then push interesting news items as they are being published. A helpful newspaper publishes a documentation on the article format, the data model of the archive, and possibly also some suggested code snippets.

2) Personalization: The traditional pull-based interaction (not obsolete) should be implemented through the system as well. This means that the *request* goes to the *in* buffer, triggers system code that fetches the appropriate file from the archive, and pushes it to the user as a reply. If that is the case, the reader may easily extend this functionality with logging of his own activity. This is useful for personalization purposes. For instance, if the reader activity bursts on a certain topic, code could be added that responds by e.g. crawling the archive for related information. If something is found, it is pushed to the reader as a suggestion.

3) Filtering: After installing a wide variety of personal code that *generates* information, some *reducing* code is needed. Code triggered by the *OutEvent* may regulate the amount of pushed messages that are allowed. This may for instance be tuned easily by using a volume control slide. This way, the suggestion mentioned in section IV-A.2 will never disturb the user, if the volume is tuned to 'quiet'.

B. Personal Filesystem

In a pervasive computing environment, where back-end services push information to a user device, the classical problem of information overload is worsened with the consumer. Even if he is able to control the flow of pushed information, he still has to decide if and where to store the received content. It is crucial that the overhead with pull-based interaction, is not simply transformed into push overhead. This would be the case if the consumer is forced to react to every push that arrives. We solve this problem by automating the receipt of pushed data items. A *Personal Filesystem*, located on a home computer, interrupts the pushed information. It has access to information on the user context, and the devices with which he is interacting. This information is used to filter pushed data in a highly personalized and context sensitive way. The prototype we are implementing is based on the general 3P structure.

1) Motivation: Future filesystems will provide better search facilities, and together with the low cost of discs, this leads to the assumption that filesystem users of tomorrow will store much larger quantities of personal information. For instance, it should be possible to store every single push that a user ever gets. It might even be a good idea, if the search facilities provide Memex[5] like functionality and finds *that interesting article on frogs* that was pushed some three years ago. This view of storage usage may lead to a situation where users want most of their pushes to go directly into the filesystem without being previously presented for the user. This is opposite to the current situation, where little is stored if the user does not do it himself. This effectively means that the user is filtering information for his filesystem. It should rather be the other way around.

2) Personal Filesystem: Using the general 3P structure leads to the following description of a *Personal Filesystem*. All pushed messages arrive at the *in* buffer. Personal code is installed to archive most of it directly, and to maintain a specific way of organizing the archive. Other code snippets specify that some message types have high importance, and should be presented for the user. These messages may be delayed, batched, collapsed or delivered immediately, depending on the priority or the user context. Since code may be triggered by *DataEvents*, it is possible to monitor the archive and push alarms when interesting situations occur, like when the buffer of delayed pushes reaches a certain limit.

Allowing filesystem functionality to be extended with user code also allows a number of activites not related to incoming data. For example, the filesystem may easily be extended with new search facilities. Installing one task to iteratively index the archive, and another, triggered by search queries, to perform index lookups.
Another example is monitoring of user interaction with the filesystem, e.g. using a text editor. If the user is writing a document on a specific topic, the filesystem may be programmed to react to this in a ways relevant to a single user only. Code working on behalf of the user in a *Personal Filesystem*, may be essential to effective management of an enormous amount of personal information.

REFERENCES

[1] Oxygen: Pervasive human-centered computing.

[2] M.K Aguilera, R.E. Strom, D.C. Sturman, M.Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[3] Tim Berners-Lee, Robert Cailliau, Jean-Francois Groff, and Bernd Pollermann. World-wide web: The information universe. *Electronic Networking: Research, Applications and Policy*, 1(2):74–82, 1992.

[4] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 1993.

[5] V. Bush. As We may Think. *Atlantic Monthly*, 176(1):641–649, January 1945.

[6] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[7] E. Freeman and D. Gelernter. Lifestreams: A storage model for personal data. *ACM SIGMOD Bulletin (ACM Special Interest Group on Management of Data)*, 25(1):80–86, March 1996.

[8] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: Toward distraction-free pervasive computing, 2002.

[9] J. Gemmell, G. Bell, R. Lueder, S. Drucker, and C. Wong. Mylifebits: fulfilling the memex vision, 2002.

[10] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A system architecture for pervasive computing. pages 177–182, September 2000.

[11] David Huynh, David Karger, and Dennis Quan. Haystack: A platform for creating, organizing and visualizing information using rdf, 2002.

[12] D. Johansen. Mobile agents: Right concept, wrong approach. January 2004.

[13] D. Johansen, K. J. Lauvset, R. van Renesse, F. B. Schneider, N. P. Sudmann, and K. Jacobsen. A tacoma retrospective. *Software Practice & Experience, Wiley*, 2002. To be published.

[14] D. Johansen, R. Van Renesse, and F. Schneider. WAIF: Web of Asynchronous Information Filters. In *Proc. of the International Workshop of Future Directions in Distributed Computing (FuDiCo)*, Betrinoro, Italy, June 2002.

[15] 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.

[16] D Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, 2000.

[17] Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.