TECHNICAL REPORT
Configurable Notifications for Event-based Systems

Eva Kühn
Space-based Compuitng Group
Argentinierstr. 8
1040 Vienna, Austria
eva@complang.tuwien.ac.at

Richard Mordinyi
Space-based Compuitng Group
Argentinierstr. 8
1040 Vienna, Austria
rm@complang.tuwien.ac.at

Christian Schreiber
Space-based Compuitng Group
Argentinierstr. 8
1040 Vienna, Austria
cs@complang.tuwien.ac.at

ABSTRACT
On the one hand publish/subscribe infrastructures, specifically notification servers, are developed to support specific application domains. On the other hand general purpose notification servers provide a large set of functionality for a broad set of application domains. Developers face the option of choosing between application specific or general purpose notification systems. Furthermore, in workflow management domains access to historic data is essential, a unique requirement in the publish/subscribe context. In this work, the extensible XVSM (extensible virtual shared memory) architecture is presented with focus on flexible notification capabilities in distributed systems demonstrated by means of the master-worker pattern.

Categories and Subject Descriptors
C.2.4 [**COMPUTER-COMMUNICATION NETWORKS**]: Distributed Systems – *distributed applications.* D.2.11 [**SOFTWARE ENGINEERING**]: Software Architectures – *Domain-specific architectures.* H.4.1 [**INFORMATION SYSTEMS APPLICATIONS**] Office Automation – *Workflow management*

General Terms
Design. Management

Keywords
Notification, distributed application, distributed shared memory, pluggable architecture

1. INTRODUCTION
The Publish/Subscribe paradigm [4] is a practical pattern for event dissemination in distributed systems. In this paradigm, a subscriber subscribes to specific kinds of events to be sent to one or more event consumers. An event producer publishes events delivered to the consumers based on the subscriptions. A central or distributed notification service (NS) can be added to decouple event producers and event consumers. The first case every client can act as a producer, consumer, or both, and connect to the central notification broker that acts as a message filtering and routing engine. This centralized way lacks of scalability and results in the problem of a single point of failure. The distributed case, multiple servers act as event brokers and cooperatively form a distributed message routing, matching and filtering engine. It acts as a messaging middleware providing asynchronous message delivery between producer and consumer and increasing flexibility and scalability. Flexibility is needed due to e.g. conflicting requirements in delivery parameters. For example, consumers may wish to filter notifications based on a set of given criteria, while producers may wish to limit the number of messages sent for reasons of system load.

A large number of publish/subscribe services have been used as the basic infrastructure for the development of distributed applications over the last few years in domains such as user and software monitoring [7], groupware applications [8], awareness tools [9], workflow management systems, and mobile applications [10]. Therefore, as detailed in [1], event notification services have to cope with requirements coming from different application domains containing the dilemma of specialization vs. generalization. Either developers use for the implementation of their distributed applications a generalized infrastructure [11], [12] supporting and integrating distributed applications but may fail to provide all the functionality needed for specific application domains, or developers install for each application domain a specialized event-based infrastructure [13], [14], [15], [16], [17], [18] and so loose the uniformity and integration of the solution. Furthermore, currently available event-based infrastructures lack of support [1] for selection and customization of the services to be provided and fail supporting mechanisms extending their functionalities.

In case of workflow management domains additional requirements [2] have to be kept in mind, which are not satisfied by traditional notification systems. One of those is the seamless access to historic data, realized in the PADRES project. It should be allowed to subscribe to data published in the future, in the past, or both requiring the need for time-based subscriptions. Time-based subscriptions provide an open or closed time-window indicating the desired set of publications. The historic data access scheme was chosen in that project due to its transparency and flexibility.

Therefore, flexibility is required both in building and maintaining notification systems with respect to the application domain and in being able to access and receive events that are being published and that have been missed by subscribers due to e.g. their previous offline status. In this work, the XVSM architecture [23], [25] is described with respect to its notification capabilities. XVSM stands for extensible virtual shared memory and aims to support coordination and collaboration of distributed systems [24], [26]. Extensibility is achieved by means of its modular design, the usage of aspects and a basic API allowing the development of notifications mechanisms taking domain specific application aspects fully into consideration.

The remainder of this paper is structured as follows: Section 2 introduces the proposed architecture. Section 3 explains the notification flavors possible with the proposed architecture while section 4 describes a use case with several different notification requirements to demonstrate them. Section 5 summarizes related work, and finally, section 6 presents conclusions and suggests directions for future work.

2. RESEARCH APPLICATION

The section will provide an introduction into the XVSM architecture; will explain its API and discusses available extension capabilities.

2.1 XVSM Architecture

XVSM stands for extensible virtual shared memory. It is a space based computing middleware which defines a Linda [3], [5] tuple space based extension.

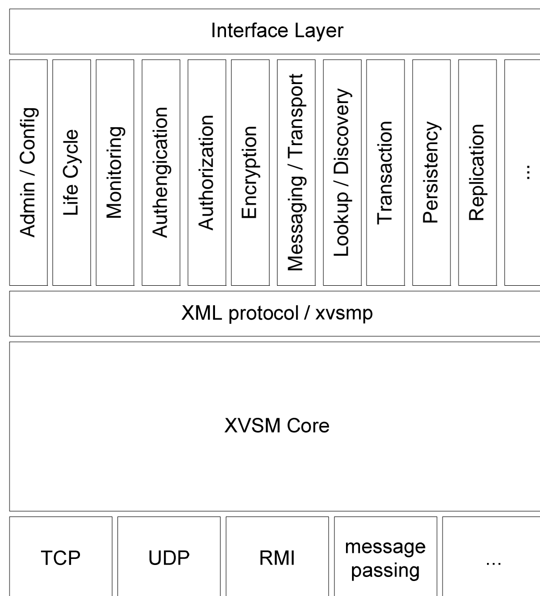

Figure 1: XVSM core architecture

The basic concept is a *container*. This is a programmable object in the space which can store and administrate so called data *entries*. A container can be addressed by its *container reference* URL ("cref") so that a container becomes accessible like any other resource in the Internet. In addition to the cref a container can have an optional name to support lookup mechanisms. Since a container stores entries like in the Linda model it can be considered as a (named) sub-space. Another important characteristic of a container is that it can be bounded or unbounded: Bounded means that within a container there is a limited number of places for entries. Entries have no identity and can either be structured Linda tuples (cf. database records), any basic data type, XML data, RDF data, or any other complex data structure.

The XVSM architecture comprises a XVSM core and so called function profiles (section 2.3). The XVSM core is designed in a way that allows those profiles to be plugged to the core and so to provide extensions for persistency, life cycle management, or replication strategies between containers (see Figure 1). The architecture comprises language bindings that are layered above the core and use the XVSM XML protocol that is basically another representation of the XVSM API (section 2.2). Currently there are language bindings for Java and .Net. Scripting language bindings such as for Python and Scheme are under development. In addition, different technologies for communication between XVSM cores can be added to support application specific communication requirements. These can be simple TCP based protocols, remote method invocation technologies or much more complex communication mechanisms or standards. XVSM is designed to run in a peer to peer manner where each client has an XVSM runtime on its machine (rich clients) or in a client/server structure where clients only send and receive XML messages (thin clients) to and from an XVSM runtime.

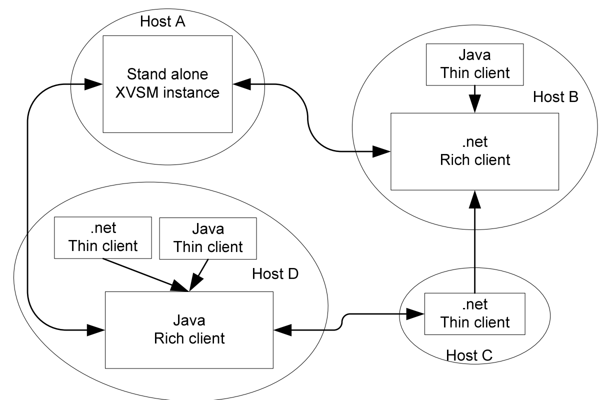

Figure 2: XVSM P2P architecture

As depicted in Figure 2 XVSM can be used as a standalone runtime (i.e. a server) which can be accessed by other instances or by thin clients. Thin clients can access standalone XVSM instances as well as instances which are running embedded within a rich client. The interoperability between the Java and .Net implementation is achieved by means of the XML protocol.

The XVSM core implements pessimistic transactions with ACID properties. A fine-grained locking mechanism is used which allows concurrent transactions. Based upon this XVSM core feature support, additional transaction-function profiles for e.g. two phase commit or distributed transactions can be implemented.

2.2 XVSM API

According to the classic Linda model, the basic operations on containers are read (read of entries), write (insertion of entries), and take (consuming read of entries). XVSM supports bulk operations as well, so that it is possible to insert multiple entries into a container resp. to read/take multiple out of within one operation. Beside the fact that read and take can be used for synchronization, as both offer a blocking behavior if no suitable entries to be read/taken are in the container, also the write operation may block. A write operation blocks if there is no place left for the entry in the container or if the same entry already exists. Therefore, the XVSM API has been extended by another operation; the so called shift that behaves like a write but instead of blocking it will replace as many entries in the container as many entries need to be written. A shift operation is a write operation that never blocks. In addition a destroy operation is supported which be-

haves like take but does not return the removed entries. Read, take and destroy are different flavors of selecting entries; they differentiate by either removing or not removing the selected entries and by returning them or not. Every operation which can block has an optional timeout parameter which is the minimum time the operation shall be retried. An operation with timeout 0 is executed exactly once and if it cannot be fulfilled it will not block. An operation with infinite timeout will wait until it can be fulfilled.

In addition to the operations on containers, operations to create and destroy containers, create, commit, rollback transactions and to add or remove aspects (see section 2.3) exist.

2.3 Extension Points

An entry can be tagged with meta information when it is inserted into a container using write or shift. For example, this meta information could be a key, a label, a priority, the position in the container, or any other user defined value. These entry tags are managed by so called *coordinators* offering a more structured coordination space. A coordinator is the programmable part of the container and is responsible for the management of the entries. XVSM distinguishes between explicit and implicit coordinators. An implicit coordinator has a view over all entries within a container (for example, coordinators which maintain an order of the entries in the container). In contrast, explicit coordinators only know those entries which had the name of the coordinator in their meta information when they were written into the container. For example, a key coordinator only knows about entries which have been written with a key. Per default, containers store entries in a random manner, according to the Linda specification. When entries are read from the container they are chosen randomly from the amount of available entries (like in JavaSpaces [19][20]). To change this behavior one can add coordinators to the container which can realize an order or a priority of entries, a complex data structure or a data management based upon the business logic of the application. Examples for these coordinators are an entry queue (first in, first out coordination), a key based management of entries or a market place based coordination which always returns the cheapest offer for a product.

Read, take and destroy use *selectors* (selectors can be considered as the “XVSM query”) to select entries. Every selector is assigned to one coordinator and can be used to query entries from this coordinator. Multiple selectors can be used within one operation. If more than one selector is used the outcome of the first selection will be used as input to the second and so on. Selectors can have a count which indicates the number of entries which have to be returned. For example, if a random selector with count 10 is used together with a key selector with value “X” the container selects ten random entries and afterwards looks in these ten entries if one of the entries is referenced by the key “X”. If an entry can be found it is returned otherwise the operation blocks until this selection can be fulfilled or the timeout of the operation expires. In addition to the meta-information based selection coordinators can be used to query based on the content or the structure of the entries. Dependent on the type of the entries stored in the container match-maker function can be used. Match-maker functions can be added dynamically, thus extending the expressiveness of the query. E.g., linda template matching [5], XML Xpath [21], RDF query mechanism [22], etc.

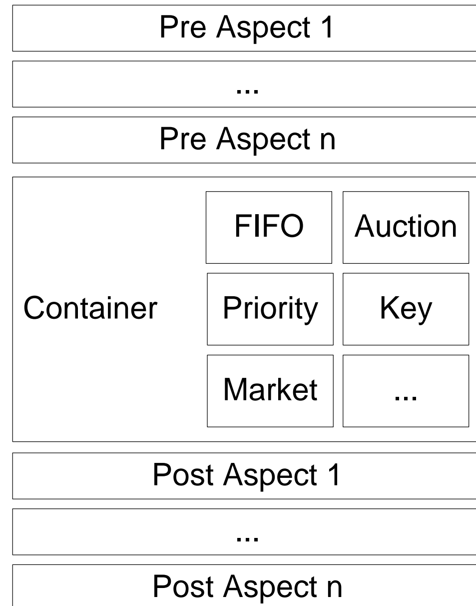

Figure 3: XVSM Container architecture

The XML protocol is designed to support a wide variety of selectors. The example in Listing 1 shows the xml representation of a key selector. In this implementation of the key coordinator every key has a name, a type and a value. This triple has to be unique within one key coordinator. In this case the count is redundant because there cannot be more than one entry with the same key/value pair. The list of properties can be customized by the implementer of the coordinator to carry any information which is necessary.

```
<selector name=”KeySelector” count=”1”>
      <properties>
            <property name=”KeyName”>
                  AKey
            </property>
            <property name=”KeyTyp”>
                  Integer
            </property>
            <property name=”KeyValue”>
                  1234
            </property>
      </properties>
</selector>
```

Listing 1: XML Representation of a key selector

Aspects can be defined on containers that in contrast to reactions in LIME [6], trigger the execution of program code upon the execution of one of the mentioned operations (read, write etc.) on a container, either before (PRE) it is called or after (POST) it has been executed. These aspects can be implemented by developers. All parameters of the operation can be accessed and modified by the developer. Using scripting languages the aspects can be configured and manipulated during the runtime. Within an aspect a developer has access to the full functionality of the XVSM API. Figure 3 depicts the XVSM Container architecture. An operation triggers all pre aspects before it enters the container. The container executes the operation according to the selectors and the meta

information. After successful execution the operation triggers the post aspects and the result of the operation is returned.

3. NOTIFICATION FLAVOURS

Figure 5 shows the general structure of processing a notification in XVSM. A publisher writes entries into a container with the name "X1". On this container an aspect is registered which intercepts this call after a successful execution and writes some information about the operation into a so called notification container. On this notification container a blocking operation waits until it can be fulfilled. When an entry which matches this blocking operation is written to the notification container the operation returns (i.e. the result of the operation is written to the answer container, see section 2.2) and the subscriber is notified.

This process is hidden behind a high level API. The subscriber only has to subscribe to the container using an appropriate notification flavor and provide a callback method which has to be executed when the notification fires.

Since the XVSM notification mechanism is based upon internal mechanisms all the mechanism described in the previous chapters can be used to create notifications. This allows a user to create a notification which exactly meets the requirements of her domain and the application.

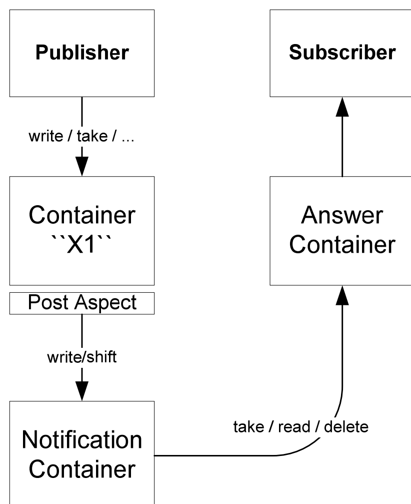

Figure 4: General structure of an XVSM Notification

In the following a detailed description of the parts of the notification is given and how they can be used to tweak the behavior of the notification.

The *notification aspect* handles the writing of the information into the notification container. The aspect can be registered for any XVSM operation (read, write …) and therefore a notification can not only be created when an entry is written. It is also possible to create notifications which notify a user when entries were read, taken or deleted. In the notification aspect any complex logic can by implemented. For example a notification shall be triggered if a specific order of messages has been detected, or if the meta information of the entry matches a specific value. In addition, one aspect can be registered at multiple containers. Therefore notifications across multiple containers can be created.

Since XVSM is a peer to peer system the *location of the involved containers* can vary. It is possible that all three containers are on the same host but also that each is on a different one. This enables the creation of durable subscriptions by putting the notification container on a host which is always reachable (for example the host where "X1" is located). The notification events are collected in the notification container whether or not the client is reachable. When the client is online again it can fetch the notifications form the container which has occurred during its absence. It also enables push and pull of notifications between peers. The answer container and the notification container can be the same container for some configuration. For example if the notification does not need to be shared between multiple clients.

The *operation* which is used to get the information from the notification container can be varied. When a client is not interested in the content of the notification but in the information that something happened a delete could be used. When the delete with the given selectors returns (an answer is written to the answer container) the client knows that a matching event occurred. With this mechanism the notification of JavaSpace technologies can be simulated [19][20]. If the number of subscribed clients is known a read operation could be used and therefore only one notification container is necessary for all the subscribers. In this case an additional aspect would be necessary which ensures that the entry is removed from the notification container after all blocking read events have been processed. If a client is only interested in a notification if a specific amount of notifications occurred a *bulk operation* can be used for retrieving the entries. This can also be used to optimize the network traffic. The number of notifications the client wants to receive can also be controlled by not renew the operation when the required amount of notification events occurred.

For each of the involved containers different *coordinators* can be used. For example if the order in which the operations occurred is important a FIFO coordinator could be used for the notification container. This ensures that even when the client is sometimes offline the notifications remain in the correct order.

Since the information which will be sent to the subscriber is selected by the notification aspect it can modified as needed (i.e. filtered, enriched with information or modified). It is also possible to notify the subscriber about *meta information* instead of the content. For example a monitoring component could be interested in the memory consumption of the XVSM instance after each operation. In this case the notification aspect would completely ignore the operation. It would read the current memory usage from the system, stores this information in an entry and writes this entry into the notification container.

When durable subscriptions are used a client could only be interested in the last 100 events which occurred during his absence. This can be realized by set the *bound of the notification container* to 100 and use to shift rather than write the notification information (the write operation would block if the container is full).

By storing the data which was altered the clients can have full access to the history of the events and to *historic data*. This can be realized by duplicating the notification container and write the notifications in both containers. From one container the notifications are taken and sent to the client. The second container contains all the historic data and a client can access this information if necessary.

4. USE CASE

In this section it is described how the master / worker pattern can be implemented to demonstrate the usage of the different forms of notifications. One component, the master, produces jobs. These jobs have to be performed by a worker. In order to get a good throughput multiple instances of the worker can be added to the scenario creating a group of workers. Normally, a job is executed by one worker only, but due to security reasons there are specific jobs which have to be executed by several workers to increase fault tolerance in case workers fail and based on majority decision make sure that only valid results are produced. This means that if three workers receive the job, the result of the calculation of a worker is only valid if at least another has produced the same result. The pattern is additionally extended by a monitoring component in order to now the ratio of open and closed jobs and a logging component to protocol any operation in the scenario. In XVSM, components communicate not directly with each other, instead communication takes place by exchanging data via containers in the XVSM space.

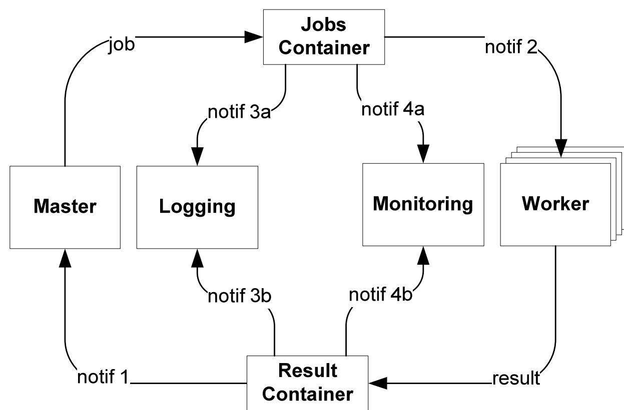

Figure 5: Master/Worker example

Figure 4 depicts the flow of information and the connection between the involved components. The master writes the jobs to be performed into a container, called Jobs Container. A job order contains a unique job identifier and all required information to execute the job. Each worker of the group of (equal) workers has a notification (notif 2) registered on that container to be informed when a new job arrives. The application scenario requires that although all of the workers are registered at the container not all of them are notified. Which workers are being notified depends on the event. The group of workers shares the same notification by accessing the same notification container. This means that workers receive the new job piggy-packed within the notification. If the workers would use read operations they would get continuously the same jobs. If the jobs would be consumed from the container (take) it would not be possible to extend the example by redundant execution of one and the same job by multiple workers to ensure a correct result. Instead the notification aspect writes as many entries into the notification container as many workers have to be notified. If the job is a non critical one only once copy is placed into the notification container. If the job has to be performed by three workers the aspect makes three copies of the job and places them into the notification container. For this scenario an additional pre-aspect has to be placed at the notification container making sure that a critical job is not consumed by the same worker several times. In addition the worker could pass some quality of service information agreements together with the registration of the notification. Based on these agreements the notification mechanism can select the worker which fits best for a specific job.

When the worker finishes the job it writes the result into the so called result container. Assuming the master is not interested in the result of the job, but it is only interested that the job has been done. For example, when the CEO of a company orders the secretary to deposit some files she might not be interested in which file cabinet the files have been deposited. The information that the files have been deposited might be enough. In such a case the notification aspect writes an acknowledgment into the notification container that is deleted by a blocking delete operation. If this happens that operation has been performed successfully notifying the client via the answer container of a successful delete operation. In any other case the Master can specify which information should be written into the notification container. Based on matching function the identifier of the job, a part of the result, or the entire result itself may be written there (the information is filtered). In addition the notification does not fire after each event that has been written into the notification container. In order to optimize the network usage and the load of the master the notification only fires when ten finished jobs arrived in the result container or a specified timeout expired. The first requirement can be achieved by performing a blocking take operation upon the notification container that has to return at least 10 matching elements. The second one extends the blocking take operation with a specified timeout. If the worker is later interested in the result of the job the result can be read from the result container based on the piggy-packed job ID. A master could also use a take operation to consume the result from the result container but with this approach the filtering of the information would have to be done by the master and the information about the result would be removed. This might be acceptable as long as only one master creates and consumes jobs. As soon as more than one master is involved the second approach would cause problems because one master could remove results which are of interest for the second master.

The logging component registers notifications on both containers (notif 3a and 3b) in order to get every incoming request and every outgoing result. This component is interested in all available information to write it into a log file. Therefore a notification is used which passes all information piggy-packed to the subscriber. No information filtering and no selectors are used. Additionally the notification is durable to ensure that when the logging component is offline the events which occurred during its absence are saved accordingly and no event is missed.

The last component involved in this scenario is the monitoring component. This component monitors the health of the system and starts or stops workers if necessary. Therefore the monitoring component holds a notification (notif 4a and 4b) on the jobs and the result container to be informed whenever something is written to the container. This notification does not contain the written entries. This component uses a notification which contains the current number of entries in the container written piggy-packed in the notification event. With this information this component can decide whether it is necessary to start additional worker or some of them can be stopped. If the number of entries in the job con-

tainer increases faster than the number of entries in the result container the monitoring component can start additional workers. In contrary, if the rate in which new entries are written to the jobs container decreases the monitoring component can stop some workers.

5. RELATED WORK

Communication in XVSM is based on the idea of shared data spaces realized in implementations like JavaSpaces [19]. In JavaSpaces registration for notification requires a template to be specified. Whenever a tuple arrives which matches the given template, an event handler is invoked. The event handler may then access the space e.g. to read or take the intended tuple. However no guarantees are made about the tuple, e.g. it may already have been taken by some other process. JavaSpaces notifications are delivered in a best effort manner. There is no guarantee at all. Events may arrive multiple times, out of order or not at all. Sequence numbers can be used by the event handler to detect possibly missed notifications. XVSM notifications are similar as in JavaSpaces. However notification about executed operations is supported. Furthermore, XVSM supports more complex queries than just a template matching. It can also take into consideration the order of entries.

The work described in [1] discusses the dilemma of specialized versus generalized notification services and introduces a configurable, extensible and dynamic architecture. The proposed XVSM architecture differs from the architecture by introducing the concepts of containers and aspects. Containers store events to allow access to historic events. Pluggable and combinable container coordinators allow not only the ordering of incoming events in any way the application domain requires, but they also provide support for application specific content matching like Linda or XPath and the search for events based on meta-information. Event sequence detection as provided in [1] can be realized by implementing and deploying a coordinator with the required functionality. Furthermore, XVSM does not specify the way (reliable, secure, …) and technology (tcp, udp, MOM, …) events are transported between XVSM instances. This completely depends on the application domain.

The PADRES project [2] uses databases to persist historic events. The XVSM architecture provides containers for event storage, which depending on the configuration of the container is able to store events either volatile or persistent. In the latter case it is achieved by means of an aspect that has a database connection established, forwarding incoming events to the database. To access historic events in PADRES, clients have to request for them explicitly, upon brokers re-publish relevant publications from their database. In contrast to this approach, in XVSM clients have full access to their containers and can independently decide how often to access and when to remove events. In PADRES upon request to republish events are sent back to the client taking the chance that the order of the events get mixed up during transmission which may be fatal in some application domains. In XVSM the order of the events is specified by the installed coordinators and do not change during transmission. Based on the used routing mechanisms in PADRES it is impossible to gain direct access to the databases. In XVSM containers can be referenced directly and so increasing flexibility with respect to application requirements. Containers may reside on the site of the publisher, on the side of the subscriber only or located at a completely different one supporting offline mode requirements of mobile applications.

6. CONCLUSION AND FUTURE WORK

In this work the XVSM architecture and its flexibility according to notification requirements in event based systems was introduced. The work presented in this paper is part of a long-term research effort. Next steps will focus on selector described execution of coordinators to provide complex subscription models and aggregation of events. Furthermore, a runtime performance evaluation compared with other notification services is targeted.

7. REFERENCES

[1] Filho, R. S., de Souza, C. R., and Redmiles, D. F. 2003. The design of a configurable, extensible and dynamic notification service. In Proceedings of the 2nd international Workshop on Distributed Event-Based Systems (San Diego, California, June 08 - 08, 2003). DEBS '03. ACM, New York, NY, 1-8. DOI= http://doi.acm.org/10.1145/966618.966633

[2] Li, G., Cheung, A., Hou, S., Hu, S., Muthusamy, V., Sherafat, R., Wun, A., Jacobsen, H., and Manovski, S. 2007. Historic data access in publish/subscribe. In Proceedings of the 2007 inaugural international Conference on Distributed Event-Based Systems (Toronto, Ontario, Canada, June 20 - 22, 2007). DEBS '07, vol. 233. ACM, New York, NY, 80-84. DOI= http://doi.acm.org/10.1145/1266894.1266908

[3] Gelernter, D. 1989. Multiple Tuple Spaces in Linda. In Proceedings of the Parallel Architectures and Languages Europe, Volume Ii: Parallel Languages (June 12 - 16, 1989). E. Odijk, M. Rem, and J. Syre, Eds. Lecture Notes In Computer Science, vol. 366. Springer-Verlag, London, 20-27.

[4] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A. 2003. The many faces of publish/subscribe. ACM Comput. Surv. 35, 2 (Jun. 2003), 114-131. DOI= http://doi.acm.org/10.1145/857076.857078

[5] Gelernter, D. 1985. Generative communication in Linda. ACM Trans. Program. Lang. Syst. 7, 1 (Jan. 1985), 80-112. DOI= http://doi.acm.org/10.1145/2363.2433

[6] Picco, G. P., Murphy, A. L., and Roman, G. 1999. LIME: Linda meets mobility. In Proceedings of the 21st international Conference on Software Engineering (Los Angeles, California, United States, May 16 - 22, 1999). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 368-377.

[7] Hilbert, D. M. and Redmiles, D. F. 1998. An approach to large-scale collection of application usage data over the Internet. In Proceedings of the 20th international Conference on Software Engineering (Kyoto, Japan, April 19 - 25, 1998). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 136-145.

[8] Dourish, P. and Bly, S. 1992. Portholes: Supporting Distributed Awareness in a Collaborative Work Group. In Proceedings of ACM Conference on Human Factors in Computing Systems (CHI '92), Monterey, California, USA, 1992.

[9] Sarma, A. and Hoek, A. v. d. Palantír: 2002. Increasing Awareness in Distributed Software Development. In Pro-

ceedings of international Workshop in Global Software Development at ICSE'2002, Orlando, Florida, 2002.

[10] Cugola, G., Di Nitto, E., and Fuggetta, A. 2001. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. IEEE Transactions on Software Engineering 27, 9 (Sep. 2001), 827-850. DOI= http://dx.doi.org/10.1109/32.950318

[11] OMG, "Notification Service Specification v1.0.1," Object Management Group, 2002.

[12] Gruber, R. E., Krishnamurthy, B. and Panagos, E. 1999. The architecture of the READY event notification service. In international Conference on Distributed Computing Systems, Workshop on Middleware, Austin, Texas, May 1999

[13] Lövstrand, L. 1991. Being selectively aware with the Khronika system. In Proceedings of the Second Conference on European Conference on Computer-Supported Cooperative Work (Amsterdam, The Netherlands, September 25 - 27, 1991). L. Bannon, M. Robinson, and K. Schmidt, Eds. ECSCW. Kluwer Academic Publishers, Norwell, MA, 265-277.

[14] Kantor, M. and Redmiles, D. 2001. Creating an Infrastructure for Ubiquitous Awareness. In Proceedings of the Eighth IFIP TC13 Conference on Human-Computer Interaction (INTERACT 2001), Tokyo, Japan, 2001.

[15] Krishnamurthy, B. and Rosenblum, D. S. 1995. Yeast: A General Purpose Event-Action System. IEEE Transactions on Software Engineering 21, 10 (Oct. 1995), 845-857. DOI= http://dx.doi.org/10.1109/32.469456

[16] Mansouri-Samani, M. and Sloman, M. 1997 GEM: A Generalised Event Monitoring Language for Distributed Systems. In Proceedings of IFIP/IEEE international Conference on Distributed Platforms (ICODP/ICDP'97), Toronto, Canada, 1997.

[17] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2001. Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems 19, 3 (Aug. 2001), 332-383. DOI= http://doi.acm.org/10.1145/380749.380767

[18] Fitzpatrick, G., Mansfield, T., Arnold, D., Phelps, T., Segall B., and Kaplan, S. 1999. Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin. In Proceedings of the Sixth European Conference on Computer Supported Cooperative Work. 1999. Copenhagen, Denmark.

[19] Freeman, E., Arnold, K., and Hupfer, S. 1999 Javaspaces Principles, Patterns, and Practice. 1st. Addison-Wesley Longman Ltd.

[20] JavaSpaces Service Specification, http://www.sun.com/software/jini/specs/js2_0.pdf

[21] XML Path Language (XPath) 2.0 W3C Recommendation, http://www.w3.org/TR/xpath20

[22] SPARQL Query Language for RDF W3C Recommendation, http://www.w3.org/TR/rdf-sparql-query/

[23] Kühn, E., Riemer, J., and Lechner, L. 2007. XVSMP/Bayeux: A Protocol for Scalable Space Based Computing in the Web. In Proceedings of the 16th IEEE international Workshops on Enabling Technologies: infrastructure For Collaborative Enterprises (June 18 - 20, 2007). WETICE. IEEE Computer Society, Washington, DC, 68-73. DOI= http://dx.doi.org/10.1109/WETICE.2007.196

[24] Mor, M., Mordinyi, R., and Riemer, J. 2007. Using Space-Based Computing for More Efficient Group Coordination and Monitoring in an Event-Based Work Management System. In Proceedings of the the Second international Conference on Availability, Reliability and Security (April 10 - 13, 2007). ARES. IEEE Computer Society, Washington, DC, 1116-1123. DOI= http://dx.doi.org/10.1109/ARES.2007.158

[25] Kühn, E., Riemer, J. and Joskowicz, G. 2005. eXtensible Extensible Virtual Shared Memory (XVSM) - Architecture and Application. Technical Report, Institute of Computer Languages, Vienna University of Technology, Austria (June 2005).

[26] Kühn, E., Riemer, J., Mordinyi, R., and Lechner, L. 2008. Integration of XVSM Spaces with the Web to Meet the Challenging Interaction Demands in Pervasive Scenarios. Ubiquitous Computing And Communication Journal (UbiCC), special issue on "Coordination in Pervasive Environments", Vol. 3, ISSN 1992-8424, March, 2008