Integrating Publish/Subscribe into a Mobile Teamwork Support Platform

S. Chaki[3] P. Fenkam[1] H. Gall[1] S. Jha[2] E. Kirda[1] H. Veith[1]

[1] Technical University Vienna
Information Systems Institute
A-1040 Vienna, Austria
[fenkam,gall,kirda]@infosys.tuwien.ac.at
veith@dbai.tuwien.ac.at

[2] University of Wisconsin
Computer Sciences Department
Madison, WI 53706
jha@cs.wisc.edu

[3] Carnegie Mellon University
Department of Computer Science
Pittsburgh PA, 15213-3891
chaki@cs.cmu.edu

Abstract

Software support for distributed and mobile collaboration is increasingly becoming a technological key issue in large and distributed enterprises. As employees are traveling or working in remote and changing locations they should be able to attend to business tasks regardless of their physical location, while using different computing devices. A requirement analysis involving case studies for two global enterprises resulted in the definition of the MOTION architecture to support distributed mobile teamwork.

In this paper we argue that publish-subscribe (P/S) is a suitable paradigm to effectively support mobile teamwork. Based on the requirements and the design principles of the MOTION architecture, we contrast the application-specific requirements for P/S components with the capabilities of existing P/S middleware. On the example of the P/S system SPEAR*, we demonstrate how P/S components can be integrated into a system that supports mobile teamwork. We present a distributed matching algorithm which combines the high performance of the* SPEAR *middleware with the expressive power of XQL subscriptions. Our results are substantiated by experimental results that demonstrate the effectiveness of our integration methodology.*

1. Introduction

Large and global organizations are increasingly faced with the problem of effectively supporting and fostering collaboration between their employees and partners. This problem is exacerbated by factors such as mobility of people and distributedness of information. For example, employees often visit departments inside or outside their organization, but still need to collaborate on different artefacts, e.g., documents and code. It is widely recognized that supporting nomadic workers is a key business requirement for large enterprises [21]. Hence, there is a growing need for a comprehensive software infrastructure which enables the employees to work together by locating information and sharing it with their collaborators.

In the European IST project MOTION (**MO**bile **T**eamwork **I**nfrastructure for **O**rganizations **N**etworks) a subgroup of the current authors contributed to the design and implementation of a platform for mobile teamwork support. The design of the MOTION infrastructure is based on the requirements elicited from two case study enterprises in the MOTION consortium. The platform for distributed and mobile collaboration is called Teamwork Services (TWS), and includes capabilities such as locating distributed business documents through peer-to-peer searches, advanced subscription and notification, community building, and mobile information sharing and access.

Publish/subscribe (P/S) is increasingly accepted as one of the most prevalent paradigms that efficiently support the construction of large scale and complex distributed software systems in general, and loosely coupled ones in particular [7, 22]. In the P/S style, components interact by subscribing to and publishing messages. Mechanisms to support the P/S style are found in commercial toolkits (e.g, Softbench [8], ToolTalk [23], TIB/Rendezvous [25]), communication standards (e.g., Corba [5]), integration frameworks (e.g., JavaBeans [11]), and programming environments. As the requirements to P/S systems are growing both with respect to performance and subscription semantics, a subgroup of the current authors have developed the prototype P/S system SPEAR [3] which allows for high expressive power in the subscription language while achieving experimentally verified high scalability. A brief background on SPEAR is provided in Section 3.

A closer look at P/S makes clear that the P/S style is suitable for communication between nomadic users: Using a P/S mechanism, a user declares interest in certain kinds of messages in terms of a *subscription*, and will receive notifications when events matching the subscription actually occur. In the context of MOTION, subscriptions may refer to events such as "a shared document has changed", "a critical piece of information has become available", or "process participant X is currently online and available for a meeting". The P/S middleware is the connector between the publishers and the subscribers. The communication between the publishers and subscribers is completely asynchronous. In particular, (i) a publisher does not need to be informed about the presence of the subscribers before sending a message, and (ii) it is always possible to plug-in a component in a system whether it is running or not. These two arguments make the P/S paradigm an intriguing candidate for mobile computing applications. The conventional concept of remote calls (RPC/RMI) is not any more adequate; for example, they require that all parties involved in a communication session need to be present before the communication can take place. It is therefore not possible to account for disconnectedness (at least, not in a straightforward manner) which however is inherent to mobile computing. We clearly see that the P/S style has a big potential to be used in collaborative settings such as MOTION.

In this paper, we investigate the integration of P/S systems into collaborative frameworks in a systematic manner, and report on our experience with integrating the P/S system SPEAR into the MOTION framework.

In Section 2, we discuss the MOTION infrastructure for supporting mobile teamwork. Using our experience from two case studies, we first distill the specific requirements for P/S systems in the context of mobile teamworks in Section 2 and 3. The requirements for the P/S system include the postulate that the language to express subscriptions be rich enough so that participants of a mobile team can construct subscriptions easily. A natural choice for the description language is the XML query language XQL. Due to performance and scalability requirements, however, existing P/S systems use less expressive languages for subscription. Being among the most expressive well-scalable P/S systems available, SPEAR uses the full expressive power of Boolean formulas as the core of its subscription language.

To bridge the semantic gap between these systems, we propose a distributed three-tier architecture in Sections 4 and 5. First, the language used by the users is translated into an intermediate language, which is designed as to be isomorphic to a fragment of XQL. The intermediate language is then mapped to the concrete language of the P/S system being integrated. Using our three-tier architecture, we have integrated the SPEAR P/S system into the MOTION architecture, and performed first practical experiments. The three-tier architecture enables us to integrate alternative P/S systems into the MOTION infrastructure easily.

The crucial step in our approach is the translation from XML/XQL to the intermediate language. This step is *approximative* to the effect that the constraints expressed

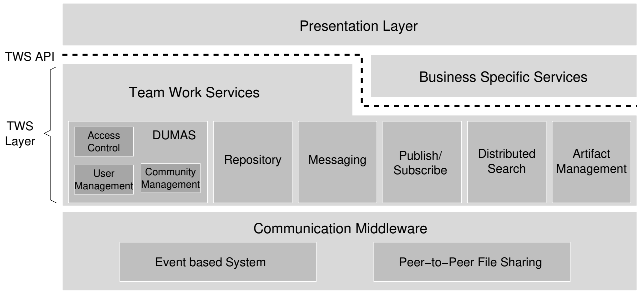

Figure 1. The MOTION Architecture

by the users' subscriptions may be weakened, and thus a small number of unwished events – which we call *negative residue events* – may be forwarded to the subscribers. To assure an overall exact matching procedure, the subscribers are equipped with a simple matching software which filters out negative residue events. Our experiments in Section 6 demonstrate that the error introduced by the approximation is indeed very small, and thus the workload incurred by our distributed matching strategy is negligible. In conclusion the combination of MOTION and SPEAR satisfies our requirements, and shows good performance in practice.

2. Services for supporting mobile teamwork

MOTION was developed to support mobile teamwork in two partner organizations. The main goals were (i) to support one of the partner organizations in locating technical documents and expertise in its widely distributed sites, and (ii) to support the other organization in organizing review meetings between mobile and nomadic users distributed around the globe.

The MOTION system typically consists of desktop computers, notebooks and PDAs all of which are called *peers*. Usually, every peer has a local repository where users can store shared documents (called *artefacts* in MOTION). In order to enable searches by other users, XML Meta-data (called *profiles* in MOTION) about each artefact is also stored. Some clients such as WAP-enabled mobile phones and Web browsers are thin clients that do not host a repository or service, but access them remotely.

The MOTION system has a three-layered architecture shown in Figure 1. The bottom layer, the communication middleware, foresees basic communication services such as publish-subscribe mechanisms, peer-to-peer file sharing and distributed searches. Previously, we have used the PeerWare [17] system as the communication middleware infrastructure which we have currently replaced by SPEAR as reported in this paper.

The middle layer in the architecture, the Teamwork Services (TWS) layer, is built using the primitives provided by the communication middleware. This layer is responsible for the integration of the main components of the system (e.g., access control, user and community management, repository). TWS provides an Application Programming Interface (API) to build services, such as storing and retrieving artefacts from the local and remote repositories, creating and managing virtual communities, and sending and receiving messages from other users. The presentation layer, the top layer of the architecture, provides the user interface to various MOTION services and is built using the TWS API. More details on the architecture developed by the MOTION consortium can be found in [12].

2.1. Publish-Subscribe in MOTION

The publish-subscribe paradigm has been identified as an architectural style that facilitates mobility and high decoupling of components [6, 26]. The publish-subscribe component of the TWS is to provide a uniform and consistent view of events to the application layer. A P/S system enables components to subscribe and receive notification of relevant events. Components can react to events by specifying callback methods that are invoked by the system whenever an event matching a subscription occurs.

We selected to use the XML query language XQL [20] as the language to express subscriptions at the presentation layer. In general, XML query languages give the business specific services the capability to query complex XML events and data. In particular, we chose XQL because it was stable at the inception of the project. Two classes of subscriptions exist in the MOTION system: system and user subscriptions. User subscriptions are initiated by users. Notification to the users are sent using various means, such as special motion messages and e-mail. System subscriptions generate events that inform components of system-specific activities, such as creation of a new MOTION user.

Based on our experience from the case studies, we have distilled the requirements for a P/S system in the context of mobile teamwork into the following list:

1. **Efficient matching mechanism.** Published events are matched with subscriptions by a P/S system. Since the number of subscriptions will be high in a mobile teamwork setting, such a matching has to scale to a large number of subscriptions.

2. **Removing subscriptions.** If users are able create subscriptions themselves, they may accumulate many subscriptions. However, some of these subscriptions will become obsolete over time. Since all subscriptions need to be matched against messages, these obsolete subscriptions will adversely effect the efficiency of the matching algorithm. Hence, there is a need to specify expiration times with subscriptions.

3. **Authorization for P/S** In collaborative applications, access restrictions and encryption mechanisms are needed to protect shared information. For example, it may not be desirable to allow every employee to subscribe to a specific event, such as an impending merger with another company.

4. **Expressive subscription language** The subscription languages provided by most existing P/S systems are often too simple to express application-level subscriptions. Subscription languages for P/S systems need to be designed that enable application developers to build more flexible and complex applications. A previous empirical attempt at using the Khronika system identified a lack of expressive subscription languages as a problematic major issue [13].

3. The SPEAR P-S Middleware

In this section we provide a short background on the P/S system SPEAR, and discuss how Spear satisfies the requirements described in Section 2.1. The SPEAR middleware implements the BDD [2] based matching algorithm presented in [3]. We will give a brief overview of the SPEAR architecture. The libraries and documentation as well as various APIs can be found at the SPEAR website.[1] The implementation of the broker in SPEAR is multi-threaded; there are four components of the broker that execute in parallel and communicate using thread-safe queues.

- The **frontend** is responsible for accepting client connections and requests.
- The **subscription updater** periodically updates the set of subscriptions based on client requests.
- The **publication matcher** matches publications against current subscriptions and generates notifications.
- The **notifier** delivers these notifications to the appropriate clients. To increase the degree of concurrency of the broker, this component is further composed of a *main notifier* and a set of *worker notifiers*.

Let us consider how well SPEAR meets the requirements described in the previous section. As demonstrated before, the BDD-based filtering algorithm used in SPEAR is very efficient [3] (this addresses requirement 1). For example, SPEAR only takes 15.57 seconds to match 1000 publications with $100{,}000$ subscriptions (see [3, Section 8]). The

[1] `http://www.cs.cmu.edu/~chaki/spear`

BDD representation of subscriptions heavily exploits the commonality between subscriptions; such commonalities are typical for subscriptions in our context. By associating expiration times with subscriptions SPEAR meets requirement 2. Moreover, the BDD representation of subscriptions lends itself to deletes in an elegant manner (see [3]) for details).

Currently, SPEAR is not equipped to address requirement 3, i.e., authorization. We are in the process of designing an authorization mechanism for SPEAR.

For performance reasons, P/S systems tend to have weak expressive power, i.e., only a limited number of properties can be formulated as subscriptions. Teamwork services oriented subscriptions, however, are more complex than well-scalable P/S systems are able to support. Thus, we need to expect a *semantic gap* between the subscription languages at the level of the TWS and the P/S middleware. We have chosen to work with SPEAR because SPEAR combines a well scalable P/S system with a relatively powerful subscription language. Nevertheless, requirement 4 is a critical issue which we will need to consider in the following sections.

4. Translation Between P/S Systems

We will now consider the translation between XQL and the SPEAR subscription language. As mentioned above, there is a semantic gap between these formalisms: A SPEAR event essentially is a list of pairs (attribute, value) where an attribute obtains at most one value per event. XML documents however, allow for repeated occurrences of the same element, cf. Listing 1. Consequently, the XML query language XQL supports existential and universal quantification. This is neither the case for SPEAR, nor for other well-scaling P/S systems. Since this particular problem is likely to occur in similar forms in other frameworks, we will consider semantic gaps in P/S systems in a systematic manner.

Let L denote a set of events, and Q a set of queries. A P/S system for L and Q is described by a total matching function $m : L \times Q \to \{\text{true}, \text{false}\}$. Thus, a P/S system is given by a triple $P = (L, Q, m)$. Let $P_a = (L_a, Q_a, m_a)$ and $P_b = (L_b, Q_b, m_b)$ be two P/S systems. A *translation* from P_a to P_b is described by two total functions $\gamma_L : L_a \to L_b$, $\gamma_Q : Q_a \to Q_b$. The translation is *exact* if

$$m_a(e,q) = \text{true} \qquad \Longleftrightarrow \qquad m_b(\gamma_L(e), \gamma_Q(q)) = \text{true}.$$

The translation is *efficient*, if γ_L and γ_Q are fast to compute, and don't increase the size of their arguments by more than a constant factor.

The main problem in many practical cases is that this theoretically appealing simple framework does not apply directly, because no well-behaved translation exists. In particular this holds true in the case of translating XML/XQL to SPEAR.

We will therefore suggest to use a *distributed matching algorithm* where the translation *approximates* a well-behaved translation in the following way:

$$m_a(e,q) = \text{true} \qquad \Longrightarrow \qquad m_b(\gamma_L(e), \gamma_Q(q)) = \text{true}.$$

The effect of such an *approximative translation* is that the peers will receive more events than they subscribed for. We will therefore need an additional matching algorithm on the individual peers which filters out the events actually subscribed for. This approach has two important consequences:

- It is necessary that the approximative translation is a good approximation, in the sense that there is a low ratio of *negative residue events* which are not excluded by the principal message broker.
- If this ratio indeed is good, then the workload for both the peer and the network are tolerable. Thus, the quality of the presented solution depends directly on the quality of the approximative translation.

5. Distributed Three-Tier Architecture

Distributed Architecture. Following the arguments of Section 4, we propose a distributed architecture that addresses the expressive gap between the top and middle tier. P/S systems are typically deployed in distributed environments. In such environments, there are subscribers, publishers, and the P/S middleware including the P/S dispatcher. The tasks of the P/S middleware include storing subscriptions, matching subscriptions against incoming events, and distributing notifications to interested subscribers, while the subscriber typically plays a passive role, subscribing for as well as receiving events.

In the previous section we discussed the principal expressive gap which arises when the required subscription language is more expressive than the languages which can be efficiently handled by the middleware. To address this expressive gap we give more responsibility to the subscribers. Instead of simply sending the subscription query to the P/S dispatcher, each subscriber also locally stores this subscription. In this new architecture, the subscription proxy stores any subscription in the local repository before sending it to the remote dispatcher. Whenever such a subscriber receives a notification from a dispatcher, it extracts the corresponding XQL query from the local repository and performs the matching. The matching performed by the subscriber is done by an exact matching algorithm; this local matching however, does not have to be highly efficient, as the rate of incoming events is low, see also Section 6.

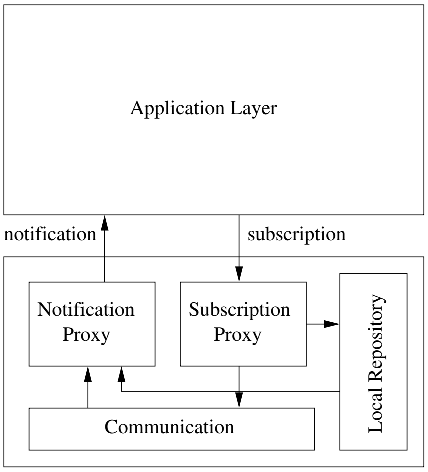

Figure 2. An Intelligent Event Based Client

Figure 2 depicts the architecture of such a subscribing peer. In contrast to this, most P/S systems such as Siena [4] and Elvin [24] have subscribers which only consist of a communication component.

Three-tier Translation. We will now describe our translation from XQL to SPEAR. We will construct this translation using an intermediate representation. We have chosen this approach to achieve greater flexibility, and facilitate future experiments with alternative P/S systems. In our software, this approach is reflected by a three-tiered architecture, where the top tier is the P/S interface on the level of XML/XQL, the intermediate layer is an abstract P/S system which we describe below, and the bottom layer is the P/S middleware, i.e., SPEAR in our case. Due to space restrictions, we will abstract from formal and syntactic details of the translation and the subscription languages, and focus on the principal ideas, supported by examples.

Top Tier. We use XML as the event language visible to the user. An example of an XML element is provided in Listing 1. Recall that XML elements are composed of other elements (called child nodes), values, and associated attributes.

```
<User>
      <Sex>'female'</Sex>
      <Address> <Zip>'1050'</Zip>
                <City>'Wien'</City></Address >
      <Address> <Zip>'5473'</Zip>
                <City>'Milano'</City></Address >
</User>
```

Listing 1: An Example of XML element

XML elements can be queried using languages such as XSL, XQL [20], XPath, XQuery [1]. We chose XQL (the XML Query language) as our subscription language at the top tier because it was the most mature language at the time the MOTION project started. However, our work can easily be extended to other query languages. XQL provides capabilities for identifying classes of nodes, boolean logic (and, or, equivalence, quantifiers), comparison of literals and vectors, casting of literals, filters, indexing into collections of nodes, methods for advanced manipulation of collections, and subscripts. XQL operators are divided into two classes: default operators and user-defined operators. Examples of default operators are comparison operators (`$gt$,$lt$,` ...) and boolean operators. The user can define additional operators, such as operators on strings.

Due to space restrictions, we cannot give a thorough introduction to XQL; Listing 2 contains a simple example of an XQL query. In XQL, one describes a subclass of nodes using notation similar to the one used to specify directories in file systems. Listing 2, line 1 refers to role elements contained in user elements. Similarly, line 2 returns user nodes which have some role but whose address has a zip code greater than 1040. We can further refine the query by stating that the user be a female (line 3-4), or a female manager, s.t. the zip codes of all her addresses are greater than 1040 (line 5-6). Note that the query now requires quantifiers.

```
1   ./User/Role
2   ./User?[./Address/Zip $gt$ 1040 ]/Role
3   ./User?[./Address/Zip $gt$ 1040 $and$
4                     Sex='female' ]/Role
5   ./User?[($all$ ./Address/Zip $gt$ 1040) $and$
6         Sex='female'  ][$any$ ./Role='manager']
```

Listing 2: Examples of XQL Queries

As mentioned above, events are XML elements, and the subscriptions are expressed by XQL queries which evaluate to true or false. In our implementation we have used the XQL engine developed by the GMD [9]. Calling this specific matching function `xqlmatch`, we obtain a P/S system (XML, XQL, `xqlmatch`) at the top tier.

The Middle Tier. The language at the middle tier is designed as an intermediate level between XML and attribute-based formalisms such as SPEAR. The intermediate event and subscription languages are called IEL and ISL (intermediate event/subscription language) respectively. XML elements are translated into IEL by collecting information at analogous XML paths in sets. Consider the transformation from Listing 1 into Listing 3. Here, both ZIPs are collected into one set, and so are both city names. Note

```
{User               |->{},
 User/Sex           |->{'female'},
 User/Address/Zip   |->{1050,5473},
 User/Address/City  |->{'Milano','Vienna'}
 User/Serialized    |->{'ptrtrtb545hr74'}}
```

Listing 3: An IEL Element

however that we lose the information that 1040 is a ZIP in Vienna. This is exactly where our translation becomes approximative. Approximation cannot be avoided, since one XML element may contain an arbitrary number of subelements (e.g. addresses) with the same name, but an attribute-based language cannot express this. IEL is intermediate in the sense that it is an attribute-based language such as the SPEAR event language SEL, but allows to associate an attribute with a set of values.

The intermediate subscription language ISL is a restricted subset of XQL which is similar in expressive power to the subscription language used by SPEAR. ISL is the fragment of XQL which does not contain quantifiers, and where all paths in the query are explicitly stated, i.e., there are no filters in the queries. Note however that ISL inherits from XQL the capability to define new operators which are not necessarily definable in SPEAR, as explained below. Predefined operators are comparison operators (such as less than and greater) and boolean operators. The abstract P/S system at the middle tier is referred to as (IEL, ISL, `islmatch`).

When translating XQL queries to approximate ISL queries, special care has to be taken for the quantifiers. We transform any universally quantified XQL subquery into a quantifier-free subquery which holds true if either the quantified attribute does not exist at all, or one of its occurrences satisfies the subquery. For instance, the XQL query `$all$ x $lt$ 5` is translated to `(($not$ x) $or$ (x $lt$ 5))`, and `$any$ x $eq$ 8` is translated to `x $eq$ 8`. Due to space restrictions, we defer a formal definition to the full version of the paper; recall however that we only have to make sure that the translation is an approximation which lets through *more* events than the original XQL subscription; the exact matching is taken care of by the subscribing peer, and the quality of the translation described is reflected by the ratio of negative residues in the experimental results.

The Bottom Tier. Our bottom tier consists of the P/S system SPEAR, given by (SEL, SSL, `spearmatch`). The subscription language SSL allows to express Boolean combinations of properties of attributes, e.g. `ZIP` gt 1004 and `city` eq “`Vienna`”. Details about the event language SEL and SSL can be found in [3].

We first show how to translate IEL to SEL. Essentially,

Operator	IEL expression	SSL expression
contains	x $contains$ ’v’	x $contains$ ’v’
starts with	x $startsw$ ’v’	x $contains$ ’#v’
equals	x eq ’v’	x$ contains$ ’#v#’
ends with	x$endsw$ ’v’	x $contains$ ’v#’
defines	x	x $contains$ ’#’

Figure 3. Mapping operators on strings.

Operator	IEL expression	SSL expression
$<$	x lt a	xmin lt a
$>$	x gt a	xmax gt a
$\leq$	x le a	xmin le a
$\geq$	x ge a	xmax ge a
$=$	x eq a	x $contains$ ’#a#’
defines	x	x $contains$ ’#’

Figure 4. Mapping numeric operators.

elements of SEL can be seen as maps from attribute names to values. On the other hand, IEL elements map attribute names to *sets* of values. We therefore encode sets of values in SEL by concatenating possible attribute values into one string. A sentinel (in our case #) is inserted between values of the set. The SEL element corresponding to the IEL element shown in Listing 3 is shown in Listing 4.

```
{ User                 |-> '##',
  User/Sex             |-> '#female#',
  User/Address/Zip     |-> '#1050##5473#',
  User/Address/Zipmin  |->  1050,
  User/Address/Zipmax  |->  5473,
  User/Address/City    |-> '#Milano##Wien#'
  User/Serialized      |->'#ptrtrtb545hr74#'}
```

Listing 4: A SEL element.

Numeric values have to be treated in a special manner. For an IEL expression such as `x |->{1,5,2,7}`, the translation algorithm also generates entries for the minimum and the maximum value in the set. Hence, `x |->{1,5,2,7}` is transformed to `{x |->#1##5##2##7#, xmin |-> 1, xmax |->7}`.

We will now describe how to translate ISL to SSL. Table 3 shows how to map operators on strings and Table 4 describes the transformation of operators on numeric values. Recall that the sentinel separating the different values is ’#’.

In the translation from (IEL, ISL, `islmatch`) to (SEL, SSL, `spearmatch`) one of the main difficulties is caused by the user defined additional operators of ISL which were inherited from XQL. Here we again make an approximative translation: Since the new operators

Total	Matched Subs.	Res.	- Res.	+ Res.
20,000	1100	15.3%	3.6%	11.8%
15,000	850	12.2%	3.2%	10.2%
10,000	520	12%	3.1%	8.9
5,000	200	6%	2.7%	3.3%

Figure 5. Residue percentages.

cannot be expressed in SPEAR directly, we replace their occurences by true, if they occur under a positive number of negations, and by false, if they occur under a negative number of negations. The effect of this construction is that the SPEAR subscription will be less constrained than the original subscription, as required in Section 4. Note however, that this strategy is only successful if the new operator does not occur too often; otherwise, it is a better strategy to extend the middleware as to handle the operator explicitly. This is foreseen in the SPEAR API, but COTS middleware may not support such an extension.

6. Experimental Results

Experiments demonstrating the general efficiency of SPEAR were provided in [3]. In this section we experimentally evaluate our distributed matching solution. Recall that in a certain sense, the load of the matching algorithm is partially distributed to the subscribers. However, we will experimentally demonstrate that the additional load on the subscriber is negligible. This is a very important issue because subscribers might be devices with limited computing power such as PDAs.

In the remainder of the paper, any event forwarded to the paper will be called a *residue*. *Positive residues* (resp. *negative residues*) are events that (resp. do not) satisfy the subscriber subscription.

For our experiment, subscriptions were generated based on the XML profiles (user, community, and artefact profiles) available in the MOTION prototype. Publications were randomly chosen among the artefact profiles. Table 5 shows the results of our experiment. The first column is the total number of subscriptions, the second column is the number of subscriptions matched against a fixed set of publications (1000 in our experiments) by SPEAR, the third column is the number of residues obtained, while the fourth and fifth columns give the number of positive and negative residues. All percentages are given relative to the number of matched subscriptions (second column).

One of the most important metrics is the ratio between the number of negative and positive residues. On average, the number of positive residues was three times the number of negative residues. This clearly demonstrates that the network traffic overhead caused by sending negative residues to the subscribers can be neglected. In our case, on average only 3.15% of the matched events were negative residues.

Suppose that the 1000 events/publications were created in a system with 1000 peers within one minute. Considering the experimental setting for the case of 20000 subscriptions, this means that each subscriber has 20 subscriptions, which is a realistic assumption. In such an environment, on average each peer receives 66 notifications per hour, of which only 2 to 3 (i.e., 3.16%) have to be discarded per hour. Thus, the workload for the peers/clients is indeed negligible, and we expect the system to scale well to higher numbers of peers.

To evaluate the real-life performance in an industrial setting, MOTION is currently being deployed by two large organizations. We continue to collect statistics on the performance of MOTION and have observed that our architecture indeed scales as the number of subscriptions and publications increase.

7. Related work

There are three general areas of related work: user awareness in collaborative environments, P/S middleware and mapping of XML-based languages to languages supported by existing P/S systems

User awareness has been identified as a key issue for collaborative tools. Various solutions exist [13, 16, 18, 19] but most of them do not explicitly tackle the mobility issue. Furthermore, they have not been designed with the goal of scaling to a large organization with thousands of employees. We believe that the use of the P/S architectural style is the key to addressing mobility and scalability.

Even though it is widely believed that the P/S paradigm is well suited for mobile computing, not all of the existing implementations meet our requirements. The event and subscription languages have to be expressive so that the users can write subscriptions easily. However, if the languages are too expressive, then the efficiency of the matching algorithm is adversely affected. Therefore, it is very important to find the right balance between expressiveness and efficiency. We consider systems at the two extremes. For instance, the subscription language of Elvin [24] is very expressive and allows all expressions that the `C` programming language allows. However, not much is known about the efficiency of the matching algorithm in Elvin. Given the expressiveness of the subscription language used in Elvin, one needs to assume that designing an efficient matching algorithm will be a challenge. Siena [4] uses a restricted set of XML, called SXML, as its language for expressing events. However, SXML is very restrictive. For instance, although events published by applications might have child nodes with the same name, only one of them is used for matching. Moreover, the fragment of XPath supported by the SXML query language neither allows filters nor quantifiers. Therefore, the languages used in Siena are not very expressive, but admit efficient matching procedures. We believe that SPEAR provides the right balance between expressiveness and efficiency. Experimental results bolster this claim.

We also proposed a distributed three-tier architecture for integrating pub-sub systems, and suggested approximative matching algorithms for an intermediate matching step. We believe that both of these are novel contributions.

8. Conclusion and Future Work

Based on a requirement analysis for publish-subscribe middleware within a system supporting mobile teamwork, we presented a three-tier architecture which enables easy integration of publish-subscribe systems into the MOTION infrastructure. To address the semantic gap between the expressive power of well-scalable publish-subscribe systems and the requirements of mobile teamwork, we presented a novel distributed two-level matching algorithm. Experimental results demonstrate clearly that the presented approach is feasible and effective for 1000 peers, and can be expected to scale up for large numbers of peers. We are currently investigating security issues of publish-subscribe systems in the context of mobile teamwork; this is the main requirement not yet addressed in the current paper.

References

[1] S. Boag, D. Chamberlin, M. Fernadez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML Query Language (XQL). Technical report, World Wide Web Consortium, April 2002. Available from http://www.w3.org/TR/xquery.

[2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[3] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient Filtering in Publish/Subscribe Systems using Binary Decision Diagrams. In *Proceedings of the 21st International Software Engineering Conference (ICSE), Toronto, Canada*, May 2001.

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

[5] The Common Object Request Broker: Architecture and specification. OMG Document Number 91.12.1, December 1991. Revision 1.1 (Draft 10).

[6] G. Cugola and E. D. Nitto. Using a Publish/Subscribe Middleware to Support Mobile Computing. In *Proceedings of the Workshop on Middleware for Mobile Computing, in association with IFIP/ACM Middleware 2001 Conference, Heidelberg, Germany*, November 2001.

[7] D. Garlan, G. Kaiser, and D. Notkin. Using tool abstraction to compose systems. *IEEE Computer*, 25(6), June 1992.

[8] C. Gerety. HP Softbench: A new generation of software development tools. Technical Report SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

[9] GMD. XQL IPSI, http://xml.darmstadt.gmd.de/xql/, 2002.

[10] C. B. Jones. *Systematic software development using VDM*. Prentice-Hall International, 1990. 2nd edition.

[11] H. Jubin. *Javabeans by example*. Upper Saddle River: Prentice Hall, 1998.

[12] E. Kirda, P. Fenkam, H. Gall, and G. Reif. A service architecture for mobile teamwork. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, July 2002.

[13] L. Lövstrand. Being selectively aware with the khronika system. In *Proceedings of the 6th European Conference on Computer Supported Cooperative Work-ECSCW'91*, September 1991.

[14] S. Microsystems. Java message service, version 1.0.2. http://www.javasoft.com, November 1999.

[15] J. Morris. *A theoretical basis for stepwise refinement and the programming calculus*. Science of Computer Programming, 1987. 2nd edition.

[16] T. Nomura, K. Hayashi, T. Hazama, and S. Gudmundson. Interlocus: Workspace configuration mechanisms for activity awareness. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work, Seattle*, pages 19–28, November 1998.

[17] G. P. Picco and G. Cugola. PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems. Technical report, Dipartimento di Electronica e Informazione, Politecnico di Milano, 2001.

[18] W. Prinz. Nessie: An awareness environment for collaborative settings. In *Proceedings of the 6th European Conference on Computer Supported Cooperative Work-ECSCW'99*, pages 391–410, September 1999.

[19] K. O. Sandor and A. Schmer. Supporting social awareness @ work, design and experience. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work, Boston*, 1996.

[20] D. Schach, J. Lapp, and J. Robie. XML Query Language(XQL). Technical report, World Wide Web Consortium, September 1998. Available from http://www.w3.org/TandS/QL/QL98/pp/qxql.html.

[21] J. Schiller. *Mobile Communications*. Addison-Wesley, Reading, Mass. and London, 2000.

[22] K. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3), July 1992.

[23] SunSoft. *Tooltalk 1.1.1 Users's Guide*, November 1993.

[24] P. Sutton, R. Arkins, and B. Segall. Supporting disconnectedness-transparent information delivery for mobile and invisible computing. In *Proceedings of 2001 IEEE International symposium on Cluster Computing and the Grid (CCGrid'01)*, May 2001.

[25] TIBCO Software Inc. TIB/Rendezvous TX Concepts Release 1.1. Technical report, TIBCO Software Inc.,Palo Alto, CA, November 2002. http://www.tibco.com.

[26] S. Zachariadis, C. Mascolo, and W. Emmerich. Exploiting Logical Mobility in Mobile Computing Middleware, July 2002.