

# Stateful Publish-Subscribe for Mobile Environments

Mihail Ionescu  
Computer Science Department,  
Rutgers University  
110 Frelinghuysen RD, Piscataway  
NJ, 08854-8058, USA  
1-732-445-3999

mihaii@caip.rutgers.edu

Ivan Marsic  
Center for Advanced Information  
Processing, Rutgers University  
96 Frelinghuysen RD, Piscataway  
NJ, 08854-8058, USA  
1-732-445-6399

marsic@caip.rutgers.edu

## ABSTRACT

The Publish-Subscribe paradigm has become an important architectural style for designing distributed systems. In the recent years, we have been witnessing an increasing demand for supporting publish-subscribe for mobile systems in wireless environments. In this paper we present SUBLIM, a stateful model for publish-subscribe systems, which is suitable for mobile systems. In our system, the server maintains a state for each client, which contains variables that describe the properties of particular clients, such as the quality of the connection or the battery utilization. The interest of each subscriber can be expressed in terms of these variables. Based on the subscriber interests, an associated agent is created on the server. The agent filters the data that reach the subscriber based on the content of the message and the current subscriber state. Experimental results show good performance and scalability of our approach.

## Keywords

Stateful publish-subscribe, run-time compiling, sublim.

## 1. INTRODUCTION

Publish-subscribe is now a well-known paradigm for building robust, large-scale distributed systems. In this architecture, the components interact via publishing messages and subscribing to classes of messages through a centralized component (the server). The set of conditions imposed by a subscriber define the subscriber's policy. A common characteristic of these architectures is that the server does not maintain any state about the subscribers. The server keeps only the address of the subscriber (usually an IP address) and the associated policy. The policy is usually specified in a language based on first-order predicates and decides whether the message is sent to the particular subscriber only based on the content of the message.

Publish-subscribe promises to be a very good paradigm for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WMASH'04, October 1, 2004, Philadelphia, Pennsylvania, USA.

Copyright 2004 ACM 1-58113-877-6/04/0010...\$5.00.

developing applications for mobile environments due to the loose coupling of the components that are involved. However, the limited and highly dynamic resources of the mobile devices (such as network connection, energy supply) impose major challenges on the way the subscribers should specify their policies. The proliferation of Wi-Fi wireless LANs makes this situation even worse, since the clients need to be able to make a smooth transition from the hotspots to areas with normal or poor coverage. The main question we address in this paper is: What type of policies do the subscribers need to specify in order to be able to receive the right information in such environments?

There has been a considerable body of research in the last years on supporting publish-subscribe in mobile environments. The main focus was to improve the performance of the overall system in the presence of sporadic connectivity, using different algorithms for finding the best path to the mobile subscriber or support for offline work. In this paper we argue that we need novel mechanisms for the deployment of the policies themselves in order to support publish-subscribe in mobile environments.

A wide range of applications use the publish-subscribe paradigm, from simple stock quotes to civil disaster recovery or customized enterprise applications. To illustrate the generality of the problems that are faced in mobile environments we will give short examples in each category.

Suppose that in a stock market application, a client wants to receive information about the companies DELL or IBM when the stock quote drops under \$30. The information about the company may contain additional data, like company name, percentage change and the number of shares that were transacted in a given period. In a mobile environment, the client would like to specify how the conditions change depending on information like the quality of the connection or battery power left. For example, the client may be interested in getting the stock quotes of IBM, DELL and Microsoft if it has a good connection and a lot of battery power left, but only in IBM if the price goes under \$20 and less than 30% battery power is available. Also, if the client has a bad connection, it might be interested in getting only the actual quote for the stock price with no additional information like percentage change and the number of shares that were transacted.

Consider now a civil disaster recovery scenario, where agents in the field are equipped with mobile devices and are searching in a specified region. They can receive periodical updates from

different data sources about any activity that was discovered in the area. The information can be a complex map or just some simple map indications. It should be possible for the agent to define its policy based on the current state it is in, depending on the network connectivity, remaining power, etc.

Finally, let us consider a large hospital with an Intensive Care Unit. The patients are monitored continuously using specialized devices. Ideally, the doctor would like to receive all of the data for the patients he is responsible for, if he has a good network connection and the battery of his PDA is charged. However, if the network connection deteriorates, he would like to be sure that he receives the most important data (such as the pulse) and maybe compressed versions of the least important data, even with loss of information such as the electrocardiogram. Moreover, if the battery power is very low, he would like only a summary of the state of the patients, at a regular interval of time.

We believe that the main characteristics that make the publish-subscribe paradigm suitable for mobile systems are the following:

- Stateful subscribers

The server must maintain a state associated with each subscriber. The state is an abstract concept and can be composed of any number of variables that describe the properties of a particular client.

- Rich language for expressing the conditions

A first-order predicate-based language which allows only interaction with the content of the message is no longer suitable for mobile systems. The language in which the conditions are written should be able to allow the interaction with the content of the message and with the state of the subscriber. It should also be able to modify the message content that is actually sent to the client according to the policy.

In this paper we present the SUBLIM system that supports publish-subscribe for mobile systems and meets the above requirements. Our approach can be used in conjunction with current systems, where the server maintains states only for some of the subscribers facilitating an incremental deployment. In our system, the servers are running on the access points of the wireless environments, ensuring a good scalability by supporting a potentially large number of subscribers with minimal overhead.

The paper is organized as follows. We first present the main contributions of this work and review related work in this area. Next, we describe our approach for supporting publish-subscribe in mobile systems. We then present a detailed example, elaborating the process of creation and deployment of the user policies. System performance is evaluated using this example application. Finally, we discuss further work and conclude the paper.

## 2. OVERVIEW AND MAJOR CONTRIBUTIONS

In our architecture, the subscribers are using wireless devices, such as PDAs. The server maintains a state for each client which

is a tuple-space and can be composed of any number of variables that describe the properties of the subscriber. Examples of such variables are the quality of the connection, the battery utilization or the device size. The policy of each subscriber will be able to filter the messages that reach the subscriber based on the content of the message and the subscriber state. We propose a domain specific language that will allow for supporting arbitrarily complex policies in a secure and efficient manner.

There are two major contributions of this paper. First, we introduce the concept of a stateful publish-subscribe system which is suitable for mobile systems, and present a new approach for specifying the subscribers policies. Also, our approach can be used in conjunction with the existing approaches for publish-subscribe systems in which server maintains the state for some of clients. Second, we present an efficient implementation of a stateful publish-subscribe system and provide a relevant example application.

## 3. RELATED WORK

There are two main types of publish-subscribe systems, as described in literature. Content-based publish-subscribe systems are intended for content distribution over a communication network. The most important content-based publish-subscribe systems that we are aware of are GRYPHON [1], SIENA [8], ELVIN [14] and KERYX [12]. In such systems, the subscription criteria filters messages based solely on their content.

The subscription languages of the existing systems are similar to each other and are usually based on the first-order predicates. The filtering algorithm in these situations can be made very efficient using techniques such as parallel search trees [1] or binary decision diagrams [2]. As a general observation, a trade-off between expressive subscription languages and highly efficient filtering engines is characteristic to these systems.

A couple of publish-subscribe systems were proposed to explicitly support the mobile clients in wireless environments. The JEDI system [5] provides a set of dispatching servers, organized in a tree to simplify the message routing. A client that joins the system is free of choosing one of the dispatching servers, and then using it to subscribe and receive event notifications. To support mobility, JEDI supports disconnection and reconnection of clients to the dispatching system by introducing two functions: move-out and move-in. More recent updates of the system [6,7] add several features such as managing link breaks in the tree and adapting routing strategy to the changes in the workload.

A later version of Elvin system [15] also deals with mobile clients that periodically disconnects by introducing the notion of a proxy that is responsible with maintaining the persistency of the events and delivering them to the clients upon reconnection.

The Toronto Publish-Subscribe system (ToPSS) [11] focuses on developing an efficient content-based publish-subscribe system for high speed event notification. They also introduce the notion of approximate matching-based, which tries to take into consideration the uncertainty in the subscriber policies (like location of the client).

A somehow different problem is attacked by Huang and Garcia-Molina in [9]. They consider the case of wireless ad-hoc networks with a dynamic topology and strive to provide a reconfigurable routing scheme that will find the best configuration of routing the notifications to the mobile clients. A similar approach is used in [4], which is also an excellent review of existing publish-subscribe systems for mobile environments.

We believe that our work is orthogonal with these efforts and can be used in conjunction with them. Our main contribution is that we provide a different mechanism to define the subscribers' policies and make them usable in the presence of the dynamic resources of the mobile clients, while maintaining a high generality of the system and good performance and scalability.

Ubiquitous computing is a fast growing area for research. Several approaches have been proposed to provide intelligent environments based on embedded devices. The EasyLiving project [3] aims at providing an intelligent environment by integrating the embedded and heterogeneous devices into a coherent user experience. They propose a new middleware called InConcert which provides asynchronous message passing, machine independent addressing and XML based messages protocols. The Event Heap approach is proposed in [10] in order to accommodate interactive workspaces (systems composed of a large number of embedded devices). In that project, an extended tuple-space Linda like model is proposed. The model can be made similar to publish-subscribe using pushing mechanisms at the server side.

While there is not a consensus among the researchers on the communication model that has to be used for ubiquitous environments, the majority of the proposed methods share a lot of similarities with publish-subscribe model. The work reported in this paper can be used as a starting point in investigating how we can use more complex policies for stateful subscribers in intelligent environments.

#### 4. STATEFUL PUBLISH-SUBSCRIBE SYSTEMS

The architecture for the SUBLIM system is based on a distributed network of servers in order to support a large number of subscribers. We assume that the subscribers are using wireless

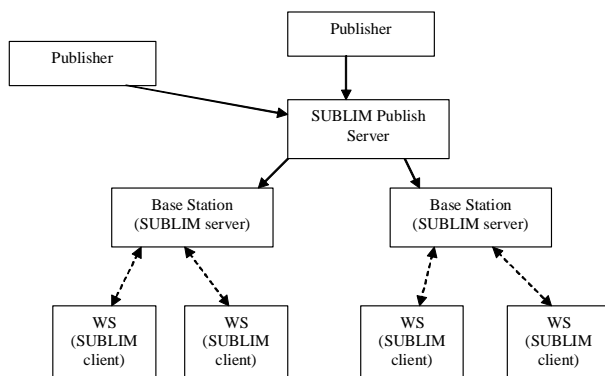


Figure 2: The SUBLIM overall architecture

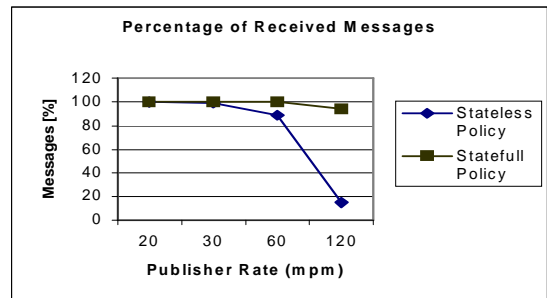


Figure 1. The architecture of the SUBLIM Server.

devices (such as PDAs). The servers are running on the access points of the wireless network, so that each server will be responsible for the subscribers connected to that access point. In Figure 1 we present the global architecture for our system, where WS stands for Wireless Subscriber. The solid lines represent wired network connections, while the dashed ones represent wireless connections. As we can notice, SUBLIM servers are running on the base stations, with wireless devices as subscribers. Publishers are also connecting to a SUBLIM Publish server that has as subscribers the servers running on the base stations. In the current implementation, the Publish Server is a simple router that forwards all of the messages to the SUBLIM Servers running on the base stations.

We will now present the abstract model used for building stateful publish-subscribe systems. The subscribers compose a community, or more specifically a S-community, where S is the SUBLIM server. To differentiate between specific subscribers, the server maintains some state for each member of the community. More formally, a S-Community is a tuple {S,C, ST, P, M}, where S is the server of the community, C is the set of subscribers, ST is a mutable set {ST<sub>x</sub> | x ∈ C} of subscribers' state, (one per each subscriber in C), P is the set of policies, (one for each subscriber in C) and M is the set of messages that can be received by the subscribers. The state is a tuple-space and can be composed of any number of variables, describing the subscriber's properties. Each variable is uniquely identified by its name. The variables can be automatic, in the sense that their values are maintained by the SUBLIM system, or manual, maintained by the policy itself.

The SUBLIM client is a component that runs on each subscriber. The SUBLIM client is responsible for maintaining the automatic variables (as detailed in Section 6.2) and dealing with the mobility (section 4.4).

When a message arrives, the arrived() event is triggered and the policy specifies the operations that have to be performed. In our current implementation the operations are specified in a C++-like syntax. In addition to all the operations supported by C++, the policy can invoke a number of system operations that are implemented by the server. The currently supported grammar is presented in Figure 3. There are three operations associated with the handling of the properties maintained in the subscriber state (getProperty, setProperty and removeProperty). The automatic properties are read only, so the setProperty and removeProperty operations will do nothing. The policy can also specify how to

create a message, using the createMessage operation, and how to manipulate the content of a message. A detailed example of such a policy is given in Section 5.

### 4.1 Server Architecture

The architecture of the server is presented in Figure 2. The subscriber creates the policy and sends it to the server using a setPolicy() command (Step 1 in Figure 2). The server generates an agent, which will be associated with this subscriber. Each time a message arrives at the server from a publisher, an arrived event is triggered at every connected participant.

The main difference in the design of the SUBLIM server compared with other publish-subscribe servers is the way the policies are written and interpreted by SUBLIM. The policy is translated into C++ and then linked together with the needed SUBLIM libraries in a dynamic link library (DLL). The server attaches the DLL and then uses the function provided by the DLL in order to make the decision. Thus, the server is very flexible and able to handle any number of heterogeneous policies in an efficient manner.

When a message m arrives (Step 2), it is processed by the Message Dispatcher, then the Subscriber Agent checks the message (Step 3) and decides if the message m' (which can be m or a modified version of m) should be sent to the subscriber or not, according to its policy (Step 4).

### 4.2 Deploying of SUBLIM

From technical implementation point of view, the servers can run on any machines accessible from wireless devices. However, we assume that the SUBLIM servers will be deployed on the base stations, so that each server will have to support a low number of subscribers, limited to the number of devices connected to one base station. This is a natural solution for wireless environments, but can constitute a limitation for mixed wired-wireless environments, where both wired and wireless subscribers need to be supported. We designed our system in order to facilitate an incremental deployment. For wired subscribers, or for subscribers that do not need a stateful policy, any of the current systems (such as Gryphon [1], or Sienna [8]) can be used. A state, and therefore the support for a stateful policy, will be maintained only for subscribers that need such advanced features (like in the example presented in Section 5), or based on special subscriptions.

### 4.3 Security and Fairness

Since the policies are actually executed on the server site on behalf of the clients, the resources of the server (in terms of memory capacity, for example) can be consumed in an unfair way by malicious or bogus policies. When the policy is translated in C++ all of the possible exceptions are caught. No system calls or pointers are allowed, so there are no possible side effects. Each client is assigned one thread, so all of the clients will receive a fair treatment. In Section 6 we will give detailed experimental results on how well the SUBLIM server is supporting bogus or malicious clients. We also implemented an asynchronous mechanism that allows the server to automatically terminate the evaluation of a policy and return a NULL message if a certain timeout expires and the policy was not evaluated. The timeout is configurable and is set in the current implementation to 500 ms.

### 4.4 Supporting Mobility

We cannot assume that one client will stay connected to one SUBLIM server for the duration of the application. The SUBLIM client monitors all the time the available base stations. When it detects that the current base station is no longer available, it will select a new SUBLIM server based on the newly available base stations. In order to maintain the current state, the SUBLIM system implements a hand-shake mechanism. Each client remembers the address of the last server it was connected to. When the client reconnects to a new SUBLIM server A, it will send in the authentication message the address of the last server B it was connected to. The server A will contact B and will retrieve the state of that particular client. Since the two servers are assumed to have a wired connection, the state is retrieved extremely fast (less than 5 ms in a LAN environment, 100 ms in a WAN), so it will be completely transparent to the client.

## 5. CASE STUDY: INTENSIVE CARE UNIT IN A LARGE HOSPITAL

Consider for example a large hospital with an Intensive Care Unit. The patients are monitored continuously using specialized devices. For simplicity, we assume that the monitoring devices generate, for each patient, the following types of data: patient ID, pulse (P) and electrocardiogram (E).

Each doctor is responsible for a certain number of patients, identified by the patient IDs. The doctor wants to receive the messages according to the following policy, denoted by P<sub>ICU</sub>.

|                                    |  |
|------------------------------------|--|
| Operations on the associated state |  |
| getProperty(name)                  | Returns the value of the specified property, or null if the property is not currently defined in the state |
| setProperty(name, value)           | Adds a property with the specified name and value  |
| removeProperty(name)               | Remove the property with the specified name from the state   |
| Operations on messages             |  |
| getData(m)                         | Get the data associated with a message   |
| setData(m,p)                       | Set the data for this message  |
| createMessage()                    | Create an empty message  |

Figure 3: The system operations

Stateless:

1. The doctor is interested in receiving all of the data (P and E) for the patients he is responsible for.

2. If the Pulse of a patient goes beyond a certain threshold, he is interested in getting all of the information for that patient, even if the patient is not his responsibility.

This policy will work very well in a wired environment, where the issues of low quality network connection, display size or energy supply do not exist. It is the thesis of this paper that, in a wireless environment, the current publish-subscribe paradigm will not be able to handle the dynamic changes in the quality of connection or energy supply. Intuitively, if many packets are sent to a client with a low bandwidth network connection, it might happen that, due to the congestion of the network, the client will not receive the most important information, (the pulse in this case) because the packets are dropped or delayed long enough to make the information irrelevant. The same argument can be made for the battery utilization. Receiving large amounts of data when the battery is low will reduce dramatically the availability of the service in real-life conditions.

Ideally, the doctor would like to receive all of the data for the patients he is responsible for, if he has a good network connection and the battery is charged. However, if the network connection deteriorates, he would like to be sure that he receives the most important data (such as the pulse) and maybe compressed versions of the least important data, even with loss of information such as the E. The same thing applies for the battery utilization. Similarly, the doctor will want to receive as much information as possible when the battery is charged, but only the most important data when the battery is low. Moreover, if the battery is very low, he would like only a summary of the state of the patients, at a regular interval of time. This leads to the following formulation (in an informal manner) of the policy  $P_{ICU}$ , suitable for mobile systems:

**R1:** The doctor is interested in receiving all data (P and E) for the patients he is responsible for if the network connection is good and the battery is charged. The electrocardiogram should not be bigger than the display size of the device used by the doctor.

**R2:** If the battery utilization goes beyond 75%, the doctor is interested in obtaining P and compressed versions of the electrocardiogram (E), with the compression ratio dependent on the battery utilization. When the battery goes beyond 25%, the doctor is only interested in the pulse (P) every five minutes.

**R3:** If the network connection deteriorates, the doctor is interested in getting the pulse P and compressed versions for the electrocardiogram (E).

**R4:** If the pulse of the patient goes beyond a certain threshold (let us say 50), the doctor is interested in getting information about that patient according to rules **R1-R3**.

The state for this policy has three automatic variables: Battery, NetworkConnection and DisplaySize. We will detail in Section 6 how the system maintains the values for these variables; for now we can assume that the values are accurate.

```
arrived(Message m, State s) :-  
/* First get the values for the automatic variables */  
1. battery=s.getProperty("Battery");  
2. networkConnection=s.getProperty("NetworkConnection");  
3. displaySize=s.getProperty("DisplaySize");  
/* Create a new message and obtain the data associated  
with the received message */  
4. Message newMessage=createMessage();  
5. pulse=m.getPulse();  
6. patientID=m.getPatientID();  
7. if (pulse > 50 and lisResponsible(patientID))  
/* The doctor is not responsible for this patient and the  
pulse is bigger than 50. Just ignore the message*/  
8. return NULL;  
9. endif  
/* If the battery is very low, send only the pulse every five  
minutes */  
10. if (battery<0.25)  
11. currentTime=getTime();  
12. oldTime=s.getProperty("Timestamp");  
13. if (oldTime==NULL)  
/* The variable does not exist. Initialize it now */  
14. s.setProperty("Timestamp",currentTime);  
15. return NULL;  
16. endif  
17. if (currentTime-oldTime>5*60)  
18. s.setProperty("Timestamp",currentTime);  
19. newMessage.setPulse(pulse);  
20. return newMessage;  
21. else  
22. return NULL;  
23. endif  
24. endif  
25. newMessage=  
compress(m,battery,displaySize,networkConnection);  
26. return newMessage;
```

**Figure 4: The implementation of  $P_{ICU}$ .**

An implementation for  $P_{ICU}$  is presented in Figure 4. Each of the rules is followed by comments (in italic) that, together with the following discussion, should provide enough understanding of the nature of our policies.

The first three lines of the policy deal with getting the current values for the automatic variables. We then create a new message and obtain the data associated with the received messages (Rules 4-5). If the doctor is not responsible for this patient and the pulse is good, the message is just discarded by returning a NULL message (again, details on how the policy determines if the patient is within the responsibility of the doctor are given in Section 6), as shown in Rules 8-9.

If the battery is very low (below 25%), only a part of the message (the pulse) will be sent to the subscriber every five minutes. This is done by using a manual variable, Timestamp, as shown in Rules 12-24.

Otherwise, the content of the message is changed based on the current values of the battery, network connection and display size. In the current implementation, the SUBLIM server provides the `compress` method, but customized methods can also be implemented in the subscribers' policies.

## 6. Implementation and performance

In this section we present some details of the current implementation in order to provide the reader with a better understanding of our architecture. We will focus on two main points: i) how the server creates the agent that will interpret the subscriber policy and ii) how the server maintains the variables in each subscriber's state, particularly the network connection.

### 6.1 Interpreting the Subscriber Policy

Upon connection to a SUBLIM server, the subscriber needs to specify its policy. The server translates the policy into C++ code, compiles it into a dynamic link library (DLL) and then loads the DLL into memory. The DLL exports only one function, which will be invoked by the arrived event. The input for this function is the message, encapsulated in a special class (Message) and the current state of the subscriber. The output of this method will be the message that it is sent to the subscriber. A NULL message means the subscriber is not interested in this message. Compiling the policy into an executable code (DLL) only once, and loading the DLL into memory significantly improves the performance of the whole system

### 6.2 Maintaining the Automatic Variables in Subscriber State

The automatic variables from the state of each client are maintained by the SUBLIM server using a simple protocol of communication with the subscriber. Periodically (30 seconds in the current implementation), the client sends to the server the new values for the variables, in a state package. Variables such as battery utilization, display size, etc. are obtained directly at the subscriber site and sent back to the SUBLIM server. However, for the network connection variable, additional computation needs to be done at the server site. The server records the number of messages sent to subscriber *S* during an interval of time, `nMessagesSent`. Subscriber *S* sends back to the server, in the state package, the number of messages it received in that interval, `nMessagesReceived`. The network connection variable is simply the ratio `nMessagesReceived/nMessagesSent`. Intuitively, a high value (close to 1) of this ratio means that the bandwidth can support the current traffic to that subscriber. If the ratio decreases, it means that the messages are lost or delayed in the network, so the subscriber policy needs to apply special measures, like the ones described in Section 5.

We cannot assume that one client will stay connected to one SUBLIM server for the duration of the application. The SUBLIM client monitors the available base stations permanently. When it detects that the current base station is no longer available, it will select a new SUBLIM server based on the newly available base stations. In order to maintain the current state, the SUBLIM system implements a hand-shake mechanism. Each client

remembers the address of the last server it was connected to. When the client reconnects to a new SUBLIM server *A*, it will send in the authentication message the address of the last server *B* it was connected to. The server *A* will contact *B* and will retrieve the state of that particular client. Since the two servers are assumed to have a wired connection, the state is retrieved extremely fast (less than 5 ms in a LAN environment, 100 ms in a WAN), so it will be completely transparent to the client.

### 6.3 Performance

The stateful model for publish-subscribe systems is a very generic and flexible model, allowing for a large variety of applications and policies. The performance of the system cannot be evaluated for a generic case, since it is affected by the specific application, the format of the data that has to be sent to the subscriber, the state of the subscriber and the policy of the subscriber.

Our current server fully implements the stateful model allowing for flexible deployment of various applications. For the performance evaluations, we will use the Intensive Care Unit example, with the policy specified in Section 5. The subscribers are running on wireless devices using PDAs and laptops. For the evaluation, we used three different types of PDAs, all of them running Windows CE 2.0: an HP Jornada 540, with 32 MB of RAM and a SH3 processor at 133 MHz, a Cassio Cassiopeia E-125 with 32 MB of RAM and a MIPS processor at 150 MHz and a Compaq iPaq with a strongARM processor at 300 MHz. The results are very similar for all of the three devices, so we will only show the results obtained using the Jornada device. The SUBLIM server is running on a dual Pentium Xeon at 2.8GHz with Linux 2.4.0 kernel. The publisher is connected directly to the SUBLIM server and can send messages at a variable rate. The electrocardiogram is a JPEG image, and the server is using the Independent Group JPEG library [13] in order to compress the image according to the policy  $P_{ICU}$ .

We will first study what is the overhead introduced by maintaining the state for each subscriber. The test scenario comprises *n* subscribers and one publisher, generating messages at a variable rate. When the server receives the first message from the publisher, it records this time as *start* time. The server records the *ending* time when it finishes processing all the messages it has received from the publisher. The duration of the experiment is determined by the difference between the end time and the start time.

We compare against a "blind" publish-subscribe system, in which there are no agents associated with the subscribers and no states maintained. Each subscriber receives all the messages sent by the publisher; the server acts just like a router in this case. As the metric we consider the duration of the experiment, as defined above and we define the overhead as:  $O = (D_{Full} - D_{Blind}) / D_{Blind} * 100$ , where  $D_{Full}$  is the duration of the experiment for the stateful system and  $D_{Blind}$  is the duration of the experiment for the "blind" publish-subscribe system. The average size of a message is 100 KB and the duration of each experiment is 5 minutes. The rate varies between 30 and 120 mpm (messages per minute). For this experiment, we run the subscribers on wired devices.

The results are shown in Figure 5 averaged over 20 executions for each case. As we can notice, the overhead is 0 even for a large number of subscribers (220), when the rate of the published

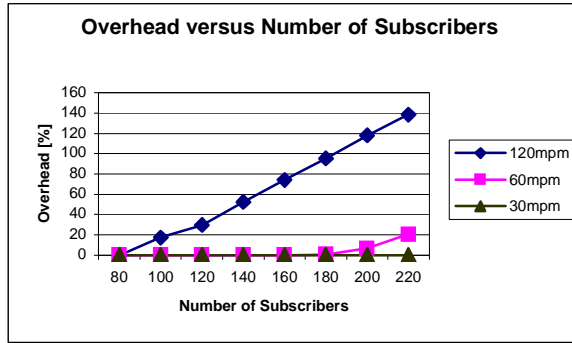


Figure 5: Overhead versus number of subscribers.

messages is not very high. However, when the rate increases to 120 mpm, the overhead becomes important when the number of subscribers is high (more than 140). This is a very promising result showing that the overhead is almost negligible, even for relatively complex stateful policies like  $P_{ICU}$ , for all practical scenarios. This is because the number of subscribers connected to one SUBLIM server is limited by the number of wireless devices connected to one base station, which is usually smaller than 100 and cannot increase due to the physical limitations of the wireless channels.

We will now present how the SUBLIM server behaves under dynamic changes of the bandwidth for a wireless subscriber. In our experiments, we have one subscriber, one patient and the publisher is able to send messages at a variable rate. We are measuring how many messages are received by the subscriber compared with the number of messages sent by the publisher, with stateful policies and stateless policies ( $P_{ICU}$  and  $P_{ICU-Stateless}$  from Section 1). The bandwidth varies randomly between 300 kbs and 3Mbs. The average size of a message is 100 KB and the duration is 5 minutes for each experiment. As in the previous experiment, the publishing rate varies between 20 and 120 mpm (messages per minute). In this experiment, the battery of the subscriber remained charged the whole time, so the number of messages that match the subscriber policy is the same in both cases. The results, averaged over 20 executions, are presented in Figure 6.

As we can notice, when the rate of the publishing messages increases, the percentage of received messages by the subscriber decreases very fast for the stateless policy. This is because the system has no way to adapt itself to the dynamic changes in the bandwidth, a well-known attribute for wireless environments. For the stateful case, the percentage is very high, even for high rates for publishing messages.

Our next measurements are related to the effect of the stateful policy on battery duration for a device. We present in Figure 7 the time needed for the battery to reach the 0 power level (the battery life), under stateful and stateless conditions. No changes in the bandwidth were performed during this experiment. The

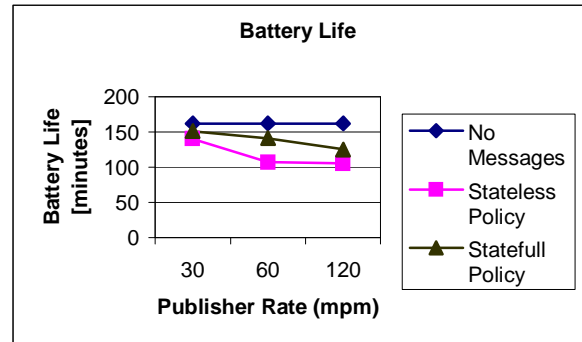
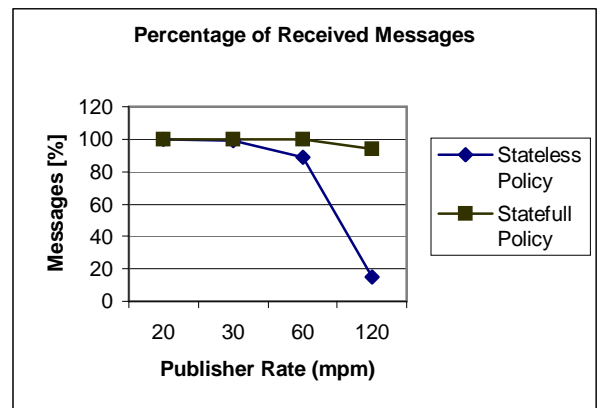


Figure 6: Battery life

first line in Figure 7 is the battery life when no messages are published. The conditions are the same as in the previous experiment: one subscriber, one publisher, one patient.

As we notice, we can extend the battery life by as much as 31%,



dramatically increasing the usability of publish-subscribe systems in wireless environments.

We finally present in Figure 8 the results for the robustness of our system in the presence of bogus/malicious agents. We use the same setup as in the first experiment, namely one SUBLIM server running on a dual Pentium Xeon at 2.8GHz with Linux 2.4.0 kernel and wired subscribers. As we specified in Section 4.3, the language used for writing the policies does not allow for use of pointers or any kind of loops, so the possibility of attacks against the server due to malicious/bogus agents is greatly reduced. However, we reduced these limitations in order to get experimental results for the robustness of the system even by allowing some subscribers to specify a policy that will contain infinite loops. A **malicious** agent is defined as an agent who sets a policy that contains an infinite loop. We were interested to see how the overhead of the system is affected by malicious agents. We basically run the same experiments as the ones from Figure 5 but we introduced a variable number of malicious agents. The results are presented in Figure 8a. We varied the percentage of malicious agents from 0 to 10. The results for the same experiments, but with a timeout value of 500 ms (as also explained in Section 4.3) are presented in Figure 8b.

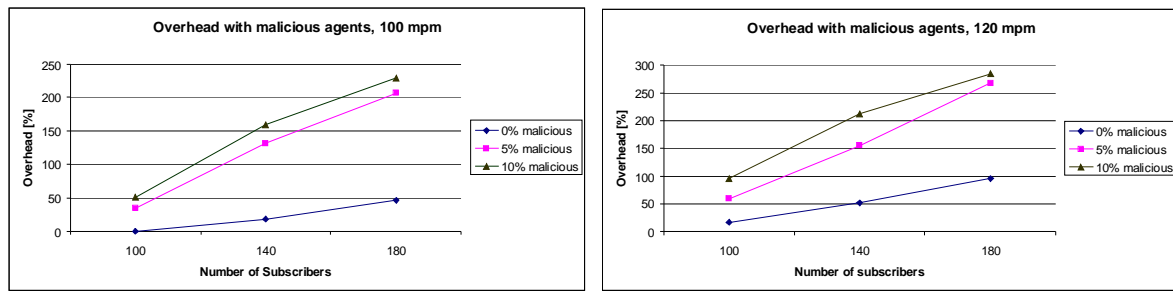


Figure 8a: Overhead with malicious agents, no timeout

| Number of subscribers | 120 mpm      |              |               | 240 mpm      |              |               |
|-----------------------|--------------|--------------|---------------|--------------|--------------|---------------|
|                       | 0% malicious | 5% malicious | 10% malicious | 0% malicious | 5% malicious | 10% malicious |
| 100                   | 17.3         | 14.6         | 13.6          | 119.66       | 118.8        | 123.33        |
| 140                   | 52.3         | 51           | 50.6          | 206.6        | 204.44       | 203.33        |
| 180                   | 95.3         | 93           | 94.6          | 288.66       | 290.55       | 287.77        |
| 200                   | 118          | 116          | 117           | 334.33       | 328          | 333.88        |

Figure 8b: Overhead with malicious agents, 500 ms timeout

As we can notice, having an asynchronous timeout mechanism is a very effective method to achieve a high robustness of the system under heavy load (200 subscribers and 240 mpm) and with a high number of malicious agents.

## 7. Conclusions

There are two major contributions of this paper. First, we introduce the notion of stateful publish-subscribe systems and argue that the current publish-subscribe systems are not suitable for mobile systems. A novel approach for expressing the conditions and building the agents using a run-time compiling technique is presented. This technique allows for the conditions to interact with the current state of each subscriber and decide whether or not to send it to the subscriber and how to modify the message if necessary.

Second, we present a motivational example where our techniques can be beneficially used. Moreover, our approach can be incorporated in the current publish-subscribe systems in an incremental manner, by allowing for some of the subscribers to maintain a state at the server side. Experimental results demonstrate the good performance, scalability and robustness of our approach.

More information about this work is provided at the web site <http://www.cs.rutgers.edu/~ionescu/Sublim>.

## 8. REFERENCES

- [1] M. K. Aguilera, R. E. Storm, D. C. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In 18th ACM Symposium on Principles of Distributing Computing (PODC), 1999.
- [2] A. Campailla, S. Chaki, E. Clarke, S. Jha and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In International Conference on Software Engineering, 2001.
- [3] B.L. Brumitt, B. Meyers, J. Krumm, A. Kern, and S. Shafer. EasyLiving: Technologies for Intelligent Environments". In Handheld and Ubiquitous Computing, 2nd Intl. Symposium, September 2000.
- [4] Mauro Caporuscio, Antonio Carzaniga and A. L. Wolf. Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications. In IEEE Transactions on Software Engineering, vol. 29, December 2003.
- [5] G. Cugola, E. Di Nitto and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. In IEEE Transactions on Software Engineering, vol. 27, pp. 827-850, Sept. 2001.
- [6] G. Cugola, G. Picco and A. Murphy. Towards dynamic reconfiguration of distributed dynamic reconfiguration of distributed publish-subscribe middleware. In 3rd International Workshop on Software Engineering and Middleware (SEM 2003), May 2002.
- [7] G. Cugola and E. Di Nitto. Using a publish-subscribe middleware to support mobile computing. In Proceedings of the Workshop on Middleware for Mobile Computing, November 2001.
- [8] A. Carzaniga, D. S. Roseblum and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In 19th ACM Symposium on Principles of Distributing Computing PODC, 2000.
- [9] Yongqiang Huang and Hector Garcia-Molina. Publish/Subscribe Tree Construction in Wireless Ad-Hoc Networks. In Proceedings of the 4th International Conference on Mobile Data Management (MDM), 2003.

- [10] Brad Johanson and Armando Fox: The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In Fourth IEEE Workshop on Mobile Computing Systems and Applications, June 2002.
- [11] H. Leung and H.-A. Jacobsen. Subject-Spaces: A state persistent programming model for publish-subscribe systems. Technical Report, University of Toronto, September 2002.
- [12] Keryx homepage, <http://keryxsoft.hpl.hp.com>
- [13] Independent JPEG group home page, <http://www.ijg.org/>
- [14] B. Segall and D. Arnold. Elvin has left the building: A publish-subscribe notification service with quenching. In Proceedings of the Australian UNIX and Open Systems User Group Conference, 1997.
- [15] P. Sutton, R. Arkins and B. Segall. Supporting Disconnectedness – Transparent Information Delivery for Mobile and Invisible Computing. In Proceedings of the IEEE International Symposium on Clustering Computing and the Grid, May 2001.