Event Notification for Mobile Clients

Örjan Berglin

Carl-Emil Lagerstedt

Examensarbete för 20 p
Institutionen för datavetenskap,
Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 20 credit points
Department of Computer Science,
Faculty of Science, Lund University

Abstract

With many Internet services, it is necessary to distribute information to clients as soon as it is available. Information should not only be delivered, it should be delivered as soon as possible. An example is weblog monitoring. As a weblog is usually updated irregularly, it is impossible to know when to check for updates. Clients do not want to waste resources on checking for new entries, they want a way to be notified when a new entry is posted. To accomplish this, we present an architecture to provide event notifications for mobile clients. In this thesis, we review some of the available systems for event notification (Siena and Elvin) before showing how XMPP (Extensible Messaging and Presence Protocol) can be used in conjunction with JEP-0060 to provide a publish/subscribe framework. We then describe our prototype, which consists of two parts: one server part, that provides a weblog service; and one client part, that uses XMPP to interact with the weblog.

Sammanfattning

För många Internettjänster är det viktigt att distribuera information till sina klienter så snart som den finns tillgänglig. Informationen ska inte bara levereras, den ska levereras så snart som möjligt. Ett exempel är bevakning av en weblog. Eftersom en weblog oftast uppdateras oregelbundet är det omöjligt att veta när man ska kontrollera om den har blivit uppdaterad. Klienter vill inte använda sina resurser till att kontrollera om ett nytt inlägg har skrivits. De vill ha ett sätt att bli aviserade när ett nytt inlägg har skrivits. För att åstadkomma detta presenterar vi en arkitektur för att skicka aviseringar till mobila klienter. I den här uppsatsen går vi igenom några av de befintliga aviseringssystemen (Siena och Elvin), varefter vi visar hur XMPP (Extensible Messaging and Presence Protocol) kan användas tillsammans med JEP-0060 för att skapa ett ramverk för publish/subscribe. Vi har utvecklat en prototyp som består av två delar: en serverdel som tillhandahåller en weblogtjänst och en klientdel, som använder XMPP för att interagera med webloggen.

Contents

1 Introduction **1**

1.1 Sony Ericsson 4

2 Use Cases **5**

2.1 Use Case: Blogging 5

2.2 Use Case: Share Picture 6

2.3 Use Case: Log Call Info 7

2.4 Use Case: Music Logging 8

2.5 Use Case: Remote Monitoring 8

2.6 Use Case: Location Logger 10

3 Publish/Subscribe Systems **11**

3.1 Events, notifications and metadata 11

3.2 Taxonomy 12

3.2.1 Notification Filtering 13

3.2.2 Subscription Types 14

3.2.3 Federation 14

3.3 Siena 15

3.4 Elvin 15

3.5 Mobility Issues in Publish/Subscribe Systems 16

4 Architecture **19**

4.1 XMPP 23

4.2 JEP-0060 25

4.3 Why XMPP? 26

5 Related Work **29**

5.1 WAP Push 29

5.2 OMA IMPS Architecture 31

5.3 IP Multimedia Subsystem 32

5.3.1 How IMS works 33

6 Security & User Integrity **35**

7 Prototype: A Blogging Service **37**

7.1 Client Software 39

7.1.1 Architecture 40

7.1.2 Java 2 Platform, Micro Edition 41

7.1.3 Smack 41

7.2 Server 42

7.2.1 The Weblog 42

7.2.2 The XMPP server 42

8 Results & Conclusions **43**

8.1 J2ME 43

8.2 OPA — Open Platform API 44

8.3 XMPP Performance 44

9 Future Work **47**

9.1 General Suggestions 47

9.2 Re-Implementation in OPA 47

9.3 Compression of XMPP Data 48

9.4 User Surveys 49

References **51**

Glossary **53**

Index **55**

List of Figures

1 Blogging 6
2 User sharing a picture 7
3 User sharing call log 8
4 Music Logging 9
5 Remote Monitoring 9
6 Location Logger 10
7 Architecture for Notification Service 20
8 Alternative Architecture for Notification Service 21
9 XMPP Overview 24
10 Establishing an XMPP Connection 25
11 WAP Push Architecture 30
12 IMS Overview 33

Acknowledgements

THERE ARE MANY people who, in one way or the other, have contributed to the making of this thesis. First of all, we would like to thank our supervisor at Sony Ericsson, David Pettersson, for his never-ending patience with us.

It is a tough job, but someone has to do it — thanks to Marcus Offesson and Stefan Johansson for reading and commenting on drafts of this thesis.

Thanks to Ovidiu Gheorghies, the author of MetaUML[1], who released a new version of MetaUML with the features we needed.

Also, thanks to Thore Husfeldt, our supervisor at the Department of Computer Science at Lund University.

I WOULD LIKE TO THANK CARIN, for supporting me and being the reason (literally) that I get up in the morning. Seriously, I could not have done it if you hadn't kicked me out of bed each morning.

//Carl-Emil

THANKS TO SOPHIA, for your patience and support. If it hadn't been for you, this thesis wouldn't be finished by now.

//Örjan

[1]MetaUML was used to draw the use case diagrams in Section 2. More information can be found at the MetaUML web site: `http://metauml.sourceforge.net`

Introduction | 1

As mobile clients get more advanced, users will expect mobile access to features hitherto available only on desktop computers. People want to read their e-mail or access web pages and files while on the move. However, the usage pattern of mobile client differs from that of stationary clients. To illustrate how such usage patterns can differ we begin with an example: blogging.

In the past few years, a new Internet phenomenon has emerged: blogging. The word blog is short for web log, and blogging is writing such a log. A blog is a web based online journal where the blog owner can publish text and readers can leave comments. Blogging applications have mostly focused on desktop users, who use their personal computer to publish and read blogs. Some effort has been made to enable mobile users to interact with web logs, but most of them provide only a rudimentary service. Currently, most mobile blogging services are used by sending SMS or MMS messages to a gateway, where the messages are processed and published. These services are targeted at the publisher of the blog, they have no functionality for reading blogs. The greatest concern with using SMS and MMS is that there is no feedback model. With SMS and MMS you get, at most, a delivery notification for the message that was sent. The SMS/MMS infrastructure does not provide any other form of notification besides the delivery notification. Some mobile phones can send e-mail, and that makes blogging trivial — there already exist e-mail interfaces for many blog services, which means that any phone that can send e-mail can also be used for blogging.

There are usually two ways to get notified when something happens on a web log. You either refresh your browser manually or you use an RSS reader that regularly check the weblog for updates. Both of these methods work by issuing HTTP GET requests to the weblog, in essence, they are user/client initiated. There is currently no way for the weblog itself to tell a user that it has been updated. This means that there might be a substantial amount of time between the time that an event occurs and the time that the user knows about it. HTTP is always client initiated, which means that the weblog has no way of telling the client to refresh her browser.

We believe that this model is not optimal for mobile users. Constant refreshing of a web page is not practical — it costs both time and money to use mobile data access, and the user might be busy doing something else. Using our proposed architecture, however, the user will simply get a message to the phone, which informs her that something has changed in one of the resources she is subscribing to. However, this is not the only advantage of our architecture. By extending the world of instant messaging we have created *instant blogging*.

This is not a problem when browsing the Internet on a PC, the user just refreshes the page to reveal updates. This is, however, not a good practice for mobile clients, since mobile users tend to have other things to consider besides surfing the net. Instead, a client will want to be notified when an update occurs. For example, if a blog is updated, a notification should be sent to any client who wants to know when the blog is updated. The biggest issue at hand is that a web server cannot take the initiative to create a connection to a client[2]; it is always the client that initiates a connection to the server. Our solution was to implement a new protocol level for event notification. On the event notification level, the client can receive notifications from the server.

The purpose of this thesis is to describe an architecture that allows notifications from the world to be sent to mobile clients. Normally, a server is not able to initiate

[2]By definition, an HTTP server is *"An application program that accepts connections in order to service requests by sending back responses" (Fielding et al., 1999)*. This means that the client has to *establish* the connection, but that the server is then free to send data over this connection without having to ask the client. More information about HTTP can be found in section **??**, *HTTP Protocol*, on page **??**.

communication with a client — all communication is initiated by the client[3](there exist push services for, for example, e-mail notification, but these services are not general).

We propose a solution where the server is able to initiate parts of the communication. We do this by requiring the client to establish a (semi)persistant connection to the server, which is used as a service connection for sending events. One way to see it is that the client initiates a session at the remote server, and then keeps the connection alive for as long as it wishes to receive events. Over this connection, the servers sends *notifications* and *event notifications*. An *event* is something that happens in the world, for example, a change of temperature, or that a web site is updated. A *notification* is a message of some kind that the recipient wants or is requested to receive. An *event notification* is a message that contains information about an event.

In this thesis, the use case that we have focused most strongly on is blogging. The reason for this is that it is a good example of a service that provides information that, more or less often, is updated. This leads to the situation where a reader of the blog can not know when to expect an update of the blog. The reader's *only* chance of kowing if the blog has been updated is to refresh her browser[4]. It is not of interest to check for updates say, every hour, because most of the times there will be no changes. The client consumes bandwidth for nothing. Then again, when there *is* an update, the user will want to review the new information as soon as possible, and not wait until the next full hour to be informed of the changes.

The architecture that we described is not confined to the blogging domain, it can be used for many types of applications. We decided to use the HTTP protocol for transport of data to and from the web server. In addition, we have implemented the XMPP protocol to send notifications and events. Both the clients and the web server are logged in to an XMPP server. A client can subscribe to events originating

[3]Note that this problem is not confined to mobile clients. All web applications have the same problem: clients can not be notified when a web page changes

[4]We consider RSS readers to be in the browser category. While they refresh contents automatically, they have no way of knowing when new content has been posted. Thus, the RSS only automates the browser refreshing.

from a certain user. When that certain user updates his blog, the web server sends an event to the event server. The event server checks which clients are subscribing to this event, and sends it to them. For us, this was an interesting challenge — could we create a new architecture layer "on top" of the web, while maintaining the current web functionality? The answer was, in the end, that, yes, we could do that.

1.1 Sony Ericsson

This thesis work was performed at Sony Ericsson in Lund. Sony Ericsson was established as a joint venture of Sony and Ericsson in 2001. Sony Ericsson's telephones are based on hardware and software from Ericsson Mobile Platforms (EMP). EMP provides an API to develop native applications for the telephone. This API is called OPA (Open Platform API). Sony Ericsson customizes the hardware platform for different models. The OPA layer is proprietary and is not accessible for third party developers. There is also J2ME support in the phones, making it possible for third party developers to create functionality.

Use Cases 2

TO FURTHER ILLUSTRATE the advantages of notification systems, we have produced some use cases that illustrate how a notification service can enrich clients. These use cases illustrate a number of different situations where traditional communication methods are not adequate for sending notifications. The use cases are both similar and different. All of the use cases deal with information that some agent wishes to publish, and that some other agent want to take part of. But the types of information are very different. However, as can be seen in the figures depicting the use cases, all of them share a common structure in publishing and subscribing to notifications.

2.1 Use Case: Blogging

Marcel has a blog that he updates rather irregularly. His friends (*Ross* and *Chandler*) are very interested in what he writes and they wish to know as soon as possible when *Marcel* has written a new blog entry. However, *Ross* and *Chandler* mostly use their cell phones to access *Marcel*'s blog. Often they access the blog, only to discover that there are no new entries. They want a way to be notified when there are new entries.

When *Marcel* writes a new blog entry, it is uploaded to a web server. The web server communicates with a notification server, that is responsible for notifying *Ross* and *Chandler* with the information that there is a new blog entry from *Marcel*.

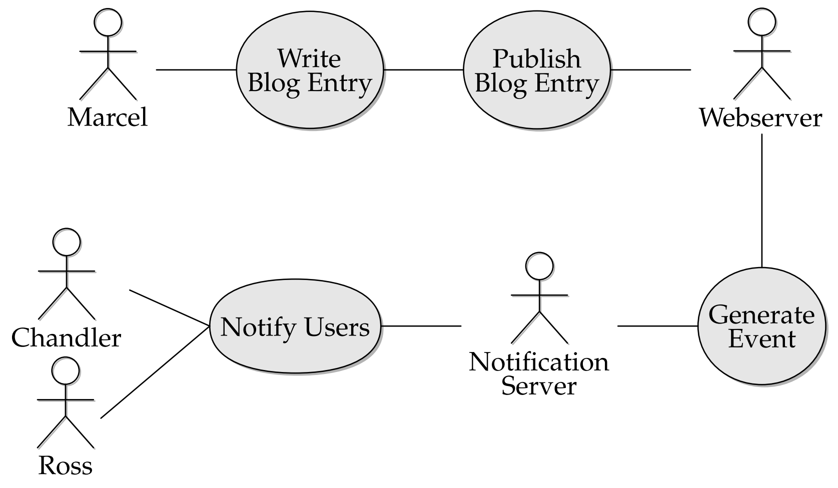

Figure 1. *Blogging*

2.2 Use Case: Share Picture

In this use case the user, *Phoebe*, wants to use her cell phone to send a picture. She also wants her friends to know that she has posted a new picture. *Phoebe* starts the photo application on the mobile client and takes a picture. This picture is sent to a web server where it is published. A notification is also created, and sent to the notification service. *Phoebe*'s friends (*Joey* and *Rachel*), who have chosen to subscribe to her shared pictures, receive notifications from the server, telling them that *Phoebe* has posted a new picture. The subscribing users can then choose to view the picture or ignore the notification.

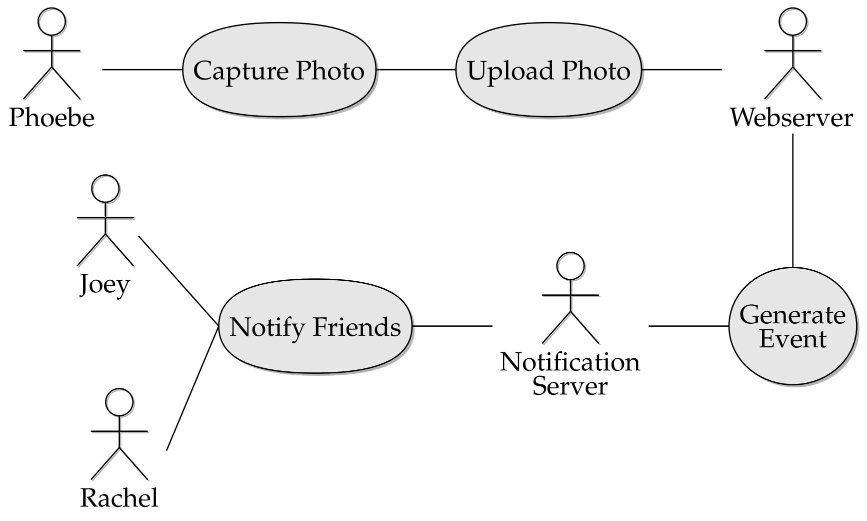

Figure 2. *User sharing a picture*

2.3 Use Case: Log Call Info

This use case describes how the device can log call information and send the information to a (password protected area of a) community. The user, *Chandler*, wishes to publish information about his phone calls on a web site. This information isn't world readable, only *Chandler* and his colleagues can read it. When *Chandler* finishes a call, the phone number, duration, and time and date for the call is published to the web site.

When *Chandler*'s coworkers receive the notification they can better plan their work, since they do not have to make calls that *Chandler* already made. Another reason to publish call info is for *Chandler* to be able to review his call history.

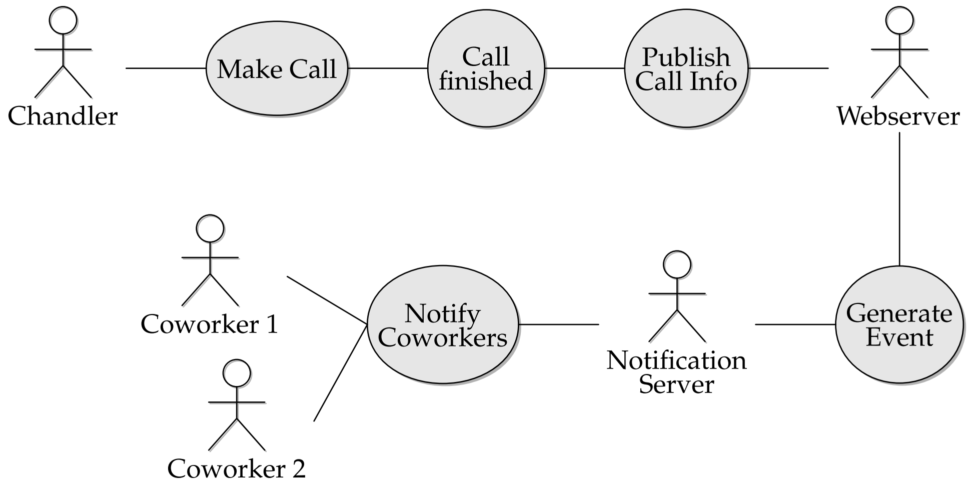

Figure 3. *User sharing call log*

2.4 Use Case: Music Logging

Here our user, *Ross*, wishes to share with the world which songs he listens to in his phone. When a song plays, the artist and title of the song is posted on a web site. The resulting list can be used to notify interested parts when a certain song is being played.

2.5 Use Case: Remote Monitoring

Suppose a chemical plant somewhere that runs an array of processes where temperatures must not exceed given values. To monitor these temperatures, an event subscription is issued. The subscription specifies which temperature events that are of interest. The publisher of these events is the thermometer unit. When an event is

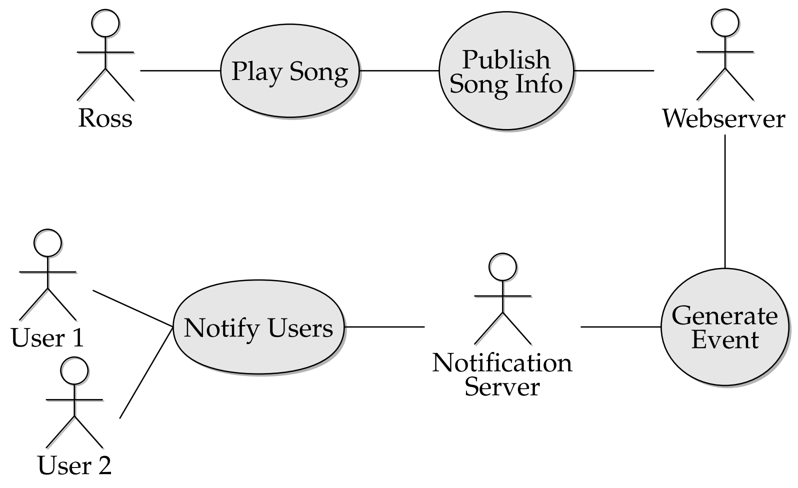

Figure 4. *Music Logging*

received, the application notifies the operator that a certain process's temperature is too high. The operator takes appropriate action to adjust the temperature. When the temperature is back to normal, a new event is issued by the thermometer unit.

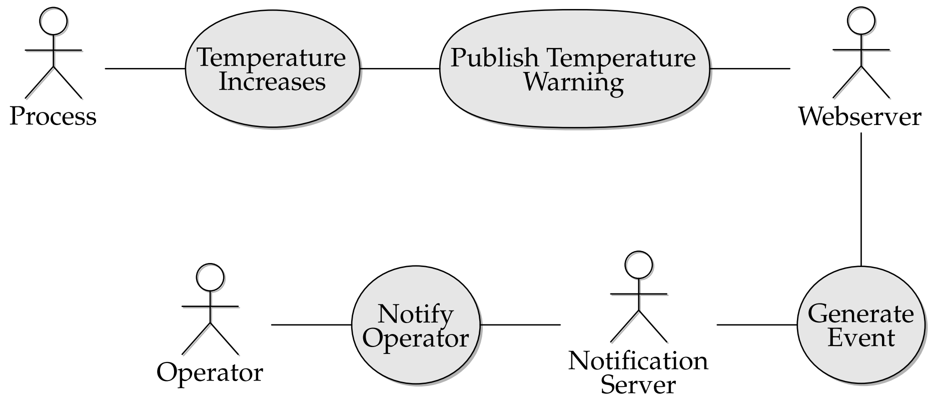

Figure 5. *Remote Monitoring*

2.6 Use Case: Location Logger

In this use case, the notification enabled application lets the user set a time interval for position logging, say 10 minutes. Then, every 10 minutes, the client's current position is transmitted as an event to the notification service. The receiving party can then use a map service, e.g. Google Local[5], to plot the user's movements on a map. This use case is yet another reason to use XMPP — there already exists a Jabber Enhancement Proposal for location, which is based on a pub/sub paradigm (Hildebrand and Saint-Andre, 2005). This enables the user to review visited locations. Another use for this is automatic tagging of pictures with location data.

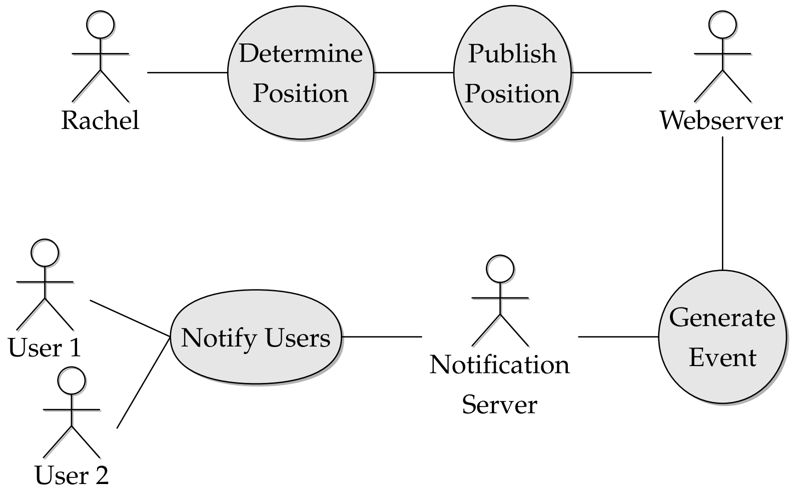

Figure 6. *Location Logger*

[5]Google Local, `http://local.google.com`, provides interactive maps and satellite photos of large parts of the world.

Publish/Subscribe Systems | 3

SOMETIMES A CLIENT needs (or wants) to be notified of events in the world. A real-world example is billing of services — normally, when a client uses some service, a bill will, sooner or later, appear in their mailbox. This is your *notification* to pay the service provider. This is a convenient method to get notified of due payments, as the user does not have to take any specific action[6] to get notified. Another method to handle payment of bills would be the constant "polling" of the service provider, constantly asking, "Do you have any bills for me to pay?"

3.1 Events, notifications and metadata

Publish/Subscribe systems (or shorter, pub/sub systems) intend to get rid of the annoyances of polling by introducing *event notifications* and *event subscriptions*.

A publisher (also called producer) is an entity that publishes events. The event consists of data that the publisher wants to make known to the world. A notification, containing meta data about the event, is propagated to all interested parts. These are denoted subscribers (also called consumers). When a notification arrives at the subscribers, they are said to have been *notified* of the event. Many schemes for handling events and notifications exist. Subscribers can subscribe to certain types of events, or simply events that contain specific content. There are many ser-

[6]Besides checking her mailbox.

vices that can be improved upon by adding event notification functionality. For example, chat services, web publishing and surveillance. In the chat service example, the chat server would send out notification messages to trigger the client's retrieval of new material. In the web publishing example, a notification would be sent out when a web page is updated. Finally, in the surveillance example, alarms that are trigggered can be sent as notifications over existing network infrastructure, eliminating the need for proprietary hardware for communication.

We continue this section with a short definition of the taxonomy of event notification services. This is followed by descriptions of some of the available systems for notification services, and a discussion of the issues that user mobility introduces.

3.2 Taxonomy

To be able to describe an event service, and to compare different services, a taxonomy is required. In this thesis, we will use the taxonomy for event services as proposed by Meier and Cahill (2005), with some additional definitions from (Carzaniga et al., 2001). The taxonomy defines three models for event notification:

- **Peer-to-peer**
- **Mediator**
- **Implicit**

In the *peer-to-peer* model, event consumers and event producers exchange events directly, without any intermediate servers. The *mediator* model has a central server where consumers subscribe to and producers publish events. The server then delivers events to subscribing parties. The consumer subscribes to events that originate from a certain user, regardless of their content. In the *implicit* event model, consumers subscribe to an *event type*. When a producer posts an event, its type is checked by the delivery subsystem and the event is delivered to interested parties. This means that a subscriber does not need to know the publisher, only the type of information that is desired.

3.2.1 Notification Filtering

Filters are used to allow the client to subscribe to desired subsets of event notifications. Every subscription can be defined as the set of events that matches a certain filter. When an event is produced, it is matched against the existing consumer filters. There are three different locations where the event filtering can be performed:

1. *producer:* in this case, the producer has a list of all the clients that has subscribed to its event notifications. This method is frugal, as the producer uses bandwidth only when matching notifications are sent. Notifications that do not match any filter are not sent. However, the producer needs access to a list of subscribers. When this list is large, or if it is frequently changed, the producer will use a lot of bandwidth just to maintain an accurate subscriber list.

2. *consumer:* this makes it easy for the receiving client to dynamically change its filter — all notifications are received by the client, but only those that match the current filter are considered. This method does not preserve bandwidth, as the producer will always send a notification to every connected client.

3. *intermediate:* in this case, the producer sends all notifications to an intermediate server. Subscribers register their subscriptions with the intermidate server. The server then delivers notifications to the clients, based on their preferences.

Meier and Cahill define four more filter models that are built by combining two or more of the above: *producer-consumer, producer-intermediate, consumer-intermediate,* and *producer-consumer-intermediate.* Of course, there could also be no filtering at all. This implies that all produced events are propagated to all of the subscribers.

3.2.2 Subscription Types

Clients use subscriptions to indicate that they are interested in certain types of events. There are three basic ways to express subscriptions (Carzaniga et al., 2001):

1. *channel-based:* the client subscribes to events that are sent over a specific channel.
2. *subject-based:* in this scheme, the client subscribes to all events that have been labeled with a specified subject, regardless of the producer. An example would be events that have been labeled with the subject "temperature".
3. *content-based:* the client's subscription is compared to the entire contents of the event. This means that we achieve a greater expressive power — events that have the "wrong" subject will still be routed to interested clients.

Most pub/sub systems use content-based subscription, as it provides the most expressive way to formulate subscriptions.

3.2.3 Federation

By federation we mean a set of interconnected notification servers (Segall and Arnold, 1997). The interconnected servers subscribe to each other's events and thus make it possible to distribute the server load and minimize latency for client events. For example, different geographical locations could have their own notification server. When a client at location A posts an event that should reach a client at location B, the federated notification servers act as pipelines for the messages. Among the different problems regarding federation is *event routing*. Traffic between the federated servers should be minimized. Therefore there must be some method to monitor the events, so that they are not sent to servers where uninterested clients are connected. However, since XMPP, which we have chosen to use for our notification service, already supports federation, we do not consider this problem.

3.3 SIENA

The SIENA event notification service, described in Carzaniga et al. (2001), is a distributed service with a set of federated servers that the clients connect to. The notification servers are organized in a peer-to-peer network. SIENA uses content-based routing. There are three basic operations[7] in SIENA:

- Publish
- Subscribe
- Advertise

A notification consists of a number of *attributes*. Every attribute has a *name*, a *data type*, and a *value*. Publishing and subscription is made by sending *publish* and *subscribe* messages, respectively. Advertisements are a special type of notifications that are used to inform subscribers about which type of content the publisher intends to broadcast. This information is used by SIENA to improve delivery of notifications. The advertisements also makes it easier for potential subscribers to decide if they are interested in notifications from a certain publisher.

3.4 Elvin

The Elvin notification service (Segall and Arnold, 1997; Segall et al., 2000), originally published by the Distributed Systems Technology Centre, University of Queensland, but now maintained by Mantara Software[8], is a notification service

[7]The *unsubscribe* and *unadvertise* operations have been omitted for brevity.
[8]`http://www.mantara.com/`

that uses content-based filtering. Elvin's service model is similar to that of SIENA. There are three basic operations[9]:

- Notify
- Subscribe
- Quenching

The *quenching* feature is an important part of Elvin. This feature makes it possible for publishers to determine if there are any clients that are subscribing to its notifications. If there are no subscribers, the publisher will simply not send these notifications. This is an efficient way to preserve bandwidth.

3.5 Mobility Issues in Publish/Subscribe Systems

There are several problems to consider when implementing pub/sub services for mobile clients. One is location — where is the client connected? We have minimized the impact of this issue by not having any federation, all notifications are handled by the same server. However, the XMPP protocol specifies how server to server communication should be done, which makes it a trivial problem when XMPP is used. Another problem is *intermittent disconnections* — we can not know, at any given time, whether the client is still connected. This is, of course, also the case when dealing with stationary clients — but they can at least be expected to disconnect gracefully in most cases, whereas mobile clients could lose connectivity at any time, due to e.g. network constraints or network outages.

We also have different constraints on the mobile client than on the stationary client. The mobile client typically has less computational power, which makes it important to make event propagation less CPU intensive. The mobile client will, most probably, have much less screen estate than the stationary client. This implies

[9] Elvin also has an *unsubscribe* function.

that the presentation will have to be adjusted for the mobile client. Finally, the space available for file storage is much less on a mobile client than it is for most stationary agents. This implies that we can not rely on the mobile client to cache a lot of data.

Architecture 4

THE SUGGESTED ARCHITECTURE, as seen in Figure 7 on the following page, consists of two layers. The first layer is the common HTTP layer. The second layer uses XMPP to transmit event notifications. The division into two layers has the advantage that clients that do not support the notification layer still can use the HTTP layer. They will, of course, not have the ability to be notified of events, but in all other aspects they can perform the same tasks as the event-enabled clients. In essence, we get a web with added notification.

To use the event layer, clients must first login to an XMPP server. This establishes an XMPP connection between the client and the server. Through this channel, notification data is passed. The client can subscribe to notifications that are generated at the notification server. For example, user *A* might subscribe to blog update event notifications generated by user *B*. When user *B* updates his blog, a notification is sent to the notification server, stating that the blog has been updated. The notification server then sends this notification to all subscribers of that event. When the clients are notified of the update, they can choose to download the new entry using HTTP (or any other transport protocol), or simply ignore it.

The notification layer can be used even with services that are not aware of the notification layer. For instance, a user might have a blog at some unknown service that has no notification capabilities.

An alternative version of the architecture is Figure 8 on page 21. Here, the web server is not aware of the XMPP server. It is the client's responsibility to produce an event and publish this event on the XMPP server. This model has the advantage

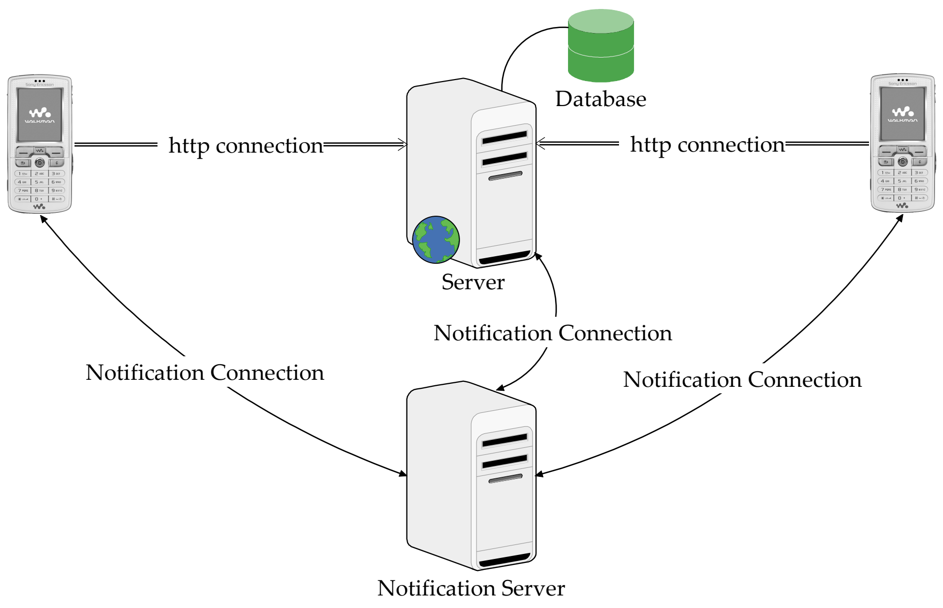

Figure 7. *Architecture for Notification Service*

that any and all services can be used with notification, since it relies on the client to provide notifications. One drawback is that the client must send more data — in addition to the posting to the web site, an XMPP message has to be composed and sent. Another drawback is that it becomes impossible to deliver notifications if the client has no notification support, the only time notifications are delivered is when the client sends one.

When a notification is produced, it should always be delivered as soon as possible. The notification server should put in a best effort to deliver notifications. Notifications are perishable, and if they are delayed they might no longer be of interest. So, the notification server's task is to get rid of (i.e. deliver) incoming notifications as quick as possible.

If, for some reason, a notification can not be delivered, the notification is stored on the notification server and is put in a queue for later delivery. Undelivered notifications have a maximum Time To Live (TTL). When the TTL is exceeded, the notification is deleted and will not be delivered to the client. This is done because otherwise a massive amount of notifications might accumulate at the server, and

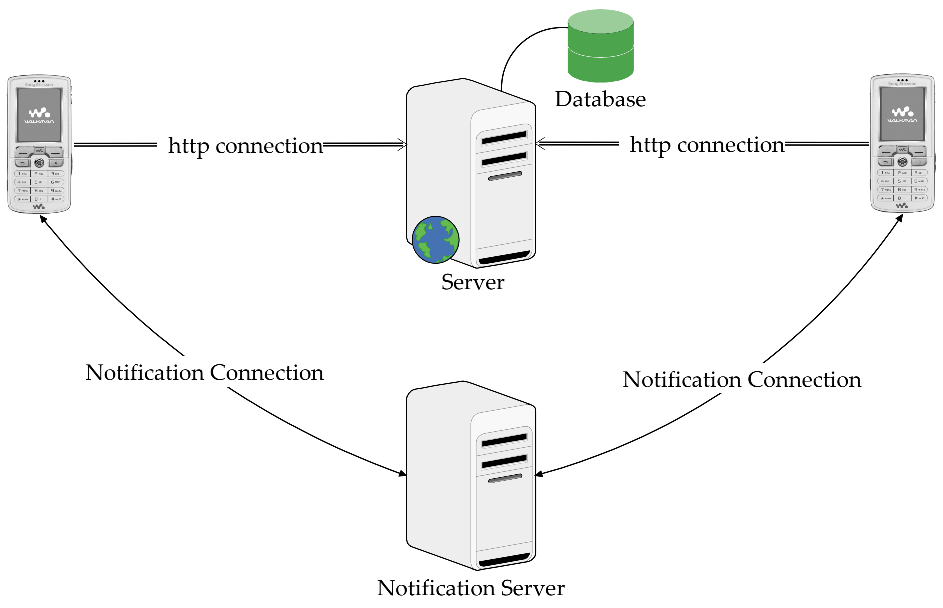

Figure 8. *Alternative Architecture for Notification Service*

these notifications are sent when the client is available again. Thus, we get notification flooding. Different types of notifications, or even notifications originating from different producers, can have different TTL's. For example, a client subscribing to a news ticker is probably not interested in news that are several days old. On the other hand, blog updates might still be interesting even after a week. There is a thin line to walk to determine the TTL of different notifications — we want the client to receive just as many notifications as he would like, but not too many. There are two main models of handling notification delivery: notifications may be delivered exactly once, or they might be delivered at most once. The latter model means that some notifications will never reach the client, while the first implies that the client really will get all the notifications that are sent to it. We suggest a mix of these two models, where some notifications that are more important are delivered according to the first model, while less critical notifications are delivered according to the second model.

The notification consist only of metadata, data about changes. For example, posting a new blog entry does not produce a notification containing the text of the entry. The notification contains information about the post, such as the name of the blog that was updated. It is then up to the subscriber to decide whether to download the data or not. This way, we keep the notification size at a minimum, and the size of each message can be predicted.

Notification filtering is done at the notification server, not at the subscribing client. The clients have, when they subscribed for notifications at the notification server, stated which notification types that they wish to receive. It is the client's own responsibility to unsubscribe to notifications that are no longer desired. Otherwise, it is assumed that the client will always want to receive subscribed notifications. The type of service to subscribe to can be on one of several different levels: 1) *client level notifications:* when a certain client publishes an event, the subscriber is notified; 2) *service level notifications:* when a certain service publishes an event, regardless of the publishing client, the subscriber is notified; and 3) *public notifications:* notifications that are transmitted to all clients. Clients can choose not to accept public notifications. Ideally, this type of notifications should be kept at lowest possible level.

The notification server does not merely pass on notifications; it may also modify notifications according to a predefined scheme. For example, mobile and stationary clients might want to receive different types of notifications for the same event. When a mobile client receives a notification, the payload should be as small as possible and only contain metadata about the event. A stationary agent, on the other hand, can receive larger notifications, that contain not only metadata, but also the event data. An example of this is picture notifications. A mobile client will receive a notification that a new picture was posted, but not the picture itself (a thumbnail version of the picture could be included). A stationary client will receive a notification that also contains the picture. One could argue that it is not the notification service's task to deliver anything except metadata, as this could put considerable load on the notification service. It is probably best to not use the notification service for delivery of anything other than metadata.

4.1 XMPP

To provide the notification service, we decided, after much consideration, to use the XMPP protocol (Saint-Andre, 2005). XMPP (Extensible Messaging and Presence Protocol) is a protocol for instant messaging, based on XML (Extensible Markup Language). XMPP was originally developed by the Jabber community and later adapted and standardised by IETF. The core parts of the protocol that provide basic functionality are described in RFC 3920 through 3923 (Saint-Andre, 2004a,b,c,d). The XMPP architecture is illustrated by Figure 9 on the following page. As its name suggests, XMPP was designed with extensibility in mind. Any extension to XMPP is specified in a JEP (Jabber Enhancement Proposal). This structure makes the protocol modular and makes it easy to add new functionality. From an implementer's point of view, the modularity means that only those parts of the protocol that are of interest for the current application needs to be implemented.

XMPP uses the concept of XML streams to enable clients to communicate through a mediating server. Specially formatted XML statements are written to the stream and are read and interpreted by the recipient. An example can be seen in Figure 10 on page 25. XMPP communication is typically done over TCP/IP, but the protocol is not bound to any particular network technology. In XMPP, a user is identified by a user name and a server name, separated by the `@` sign, for example `user@XMPPServer.com`. The user identity is denoted *XMPP ID*.

The canonical way of describing how XMPP messages are exchanged is by introducing two users; `juliet@capulet.com` and `romeo@montague.net`. When Juliet wants to chat with Romeo, her XMPP client sends her message to the XMPP server at `capulet.com`, which relays the message to the XMPP server at `montague.net`. The receiving XMPP server delivers the message to Romeo, if he is online, or saves the message for later delivery.

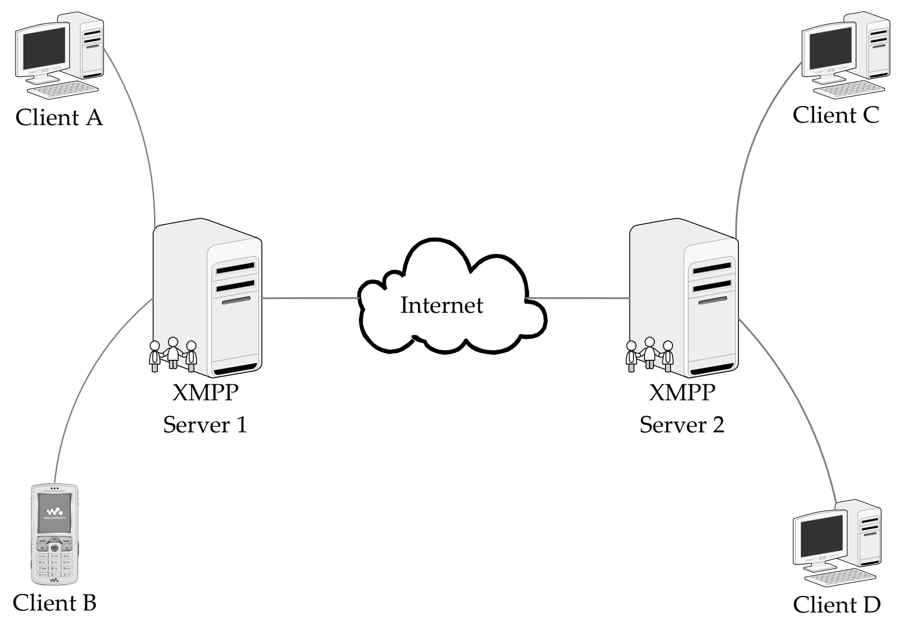

Figure 9. *XMPP Overview*

Since XMPP was not designed to preserve bandwith and XML is resource intense to parse, it could be argued that using a specialized and more compact protocol, such as Binary XML, for notification would be more efficient. This is a valid remark, as mobile data services are expensive, and cell phones typically encompasses only a fraction of the computing resources found in a desktop PC. But the bandwith issue is most likely going away with time. Current UMTS technology offers data rates of up to 384 kbit/s which is already more than enough for XMPP. Coming generations of mobile networks will provide speeds of several mbits/s. Although we could do better to preserve bandwith, we do the world a favor by not introducing a new protocol.

The same logic applies to computing power. Current medium priced handsets uses CPUs that are about as fast as a Pentium, this is already enough for our needs. In a couple of years this will also be true for the cheapest of handsets. Considering this, XML, and particulary XMPP, has many attractive features. First, it is simple and self documenting, hence easy to understand. This makes it easy for implementors to adopt. Second, since all XML-based languages are structured in the

Client:

```
<?xml version="1.0"?>
<stream:stream
  to="example.com"
  xmlns="jabber:client"
  xmlns:stream="http://etherx.jabber.org/streams"
  version="1.0">
```

Server:

```
<?xml version="1.0"?>
<stream:stream
  from="example.com"
  id="someid"
  xmlns="jabber:client"
  xmlns:stream="http://etherx.jabber.org/streams"
  version="1.0">
```

Figure 10. *Establishing an XMPP Connection*

same way, general XML-parsers and serializers can be used, which reduces the effort that goes into supporting the protocol. Third, XMPP already defines a generic publish/subscribe framework as a JEP, namely JEP-0060 (Millard et al., 2005).

4.2 JEP-0060

According to JEP-0060, an event is either *persistent* or *transient*. It is also either a *pure notification* or a notification that has a *payload*.

A persistent notification will be saved for later transmission if the subscriber is currently disconnected. Transient notifications will not be saved for later delivery. A typical example of a transient notification is a real-time stock ticker — it makes no sense to display notifications that are several hours, or even days, old. An update to a blog, on the other hand, might be of interest to a subscriber even if the update was made days ago. In this case a persistent notification should be used.

Another reason to make the distinction between persistent and transient notifications is to prevent notification buildup at the server. If all notifications were persistent, it could potentially lead to storage problems at the server.

When a publisher creates an event, it is either a pure notification or a notification with a payload. In the case of pure notification, the event is transferred unchanged to the subscribers. If the notification has a payload, the subscriber's settings decide whether the server will deliver the payload or not. This makes XMPP with JEP-0060 the perfect candidate for implementing publish/subscribe services in mobile clients, since payload download is user initiated.

4.3 Why XMPP?

There are several reasons why we have chosen XMPP as the preferred protocol. The first is that it is *standardized* by the IETF, which means that there is a common agreement on how the protocol should be implemented. This makes it easy to integrate XMPP with *many different clients*. XMPP is also *free*, in two aspects: there is no license fee to pay for using XMPP. Also, full protocol documentation is available (as opposed to proprietary protocols, the use of which usually means either paying license fees or reverse engineering the protocol, since the specifications has not been made public). The final reason for using XMPP is that major corporations are beginning to add XMPP support in their products. Programs such as Google Talk[10] and Apple's iChat[11] have the capability to exchange messages with other XMPP enabled clients. We believe that in the future, much as there is but one protocol for communicating via the World Wide Web, there will be only one protocol for instant messaging. If that protocol will be XMPP, only time can tell,

[10] `http://www.google.com/talk/`

[11] `http://www.apple.com/macosx/features/ichat/`

but it is most certainly a top candidate.

XMPP is also well suited for communication in M2M (Machine to Machine) applications.

For all the advantages of XMPP, there are also some drawbacks. For example, XML is not bandwidth preserving, which means that when many clients subscribe to the same events, there might be considerable traffic peaks.

We did consider some other protocols. One approach would have to go the way of OMA IMPS — this would mean using Binary XML. We considered this protocol to be rather limiting. There is no way to extend the protocol and still comply with the standard. Also, there are few clients that actually implements OMA IMPS. Another approach would have to develop a proprietary binary protocol. This would have the benefit of being very efficient, since we would be able to customize the protocol to our needs. In the end we decided that the effort to develop a new protocol was not worth the benefits. There is another obvious drawback — even fewer clients would use our protocol than OMA IMPS.

This led to the conclusion that we should stick to XMPP, which is already standardized and in widespread use.

Related Work | 5

In this section, we describe some of the current technologies for delivering content to mobile clients. We begin with an overview of WAP Push and continue with overviews of the OMA Instant Messaging and Presence Service, and the IP Multimedia Subsystem.

5.1 WAP Push

One might ask why we have decided to create a new architecture for notification — after all, we do have the WAP 2.0 Push framework, which is designed to do pretty much the same thing that we want to do (Pashtan, 2005). However, WAP Push has some limitations that our architecture has not. To initiate push in WAP 2.0, an HTTP POST message is sent to the client. This implies that the client acts as an HTTP server. The POST message contains a short message and an URL. The text is presented to the user who can choose to visit the accompanying URL. The WAP architecture is outlined in Figure 11 on the following page.

WAP has two different push services: *service indication* and *service load*. When content push is initiated, the push server sends and URI for the push content to the client. The client retrieves the content by issuing an HTTP GET request (in WAP 1.x, a WSP GET request is issued). When the push message is of the type *service indication*, the user is shown a short description of the push message, and

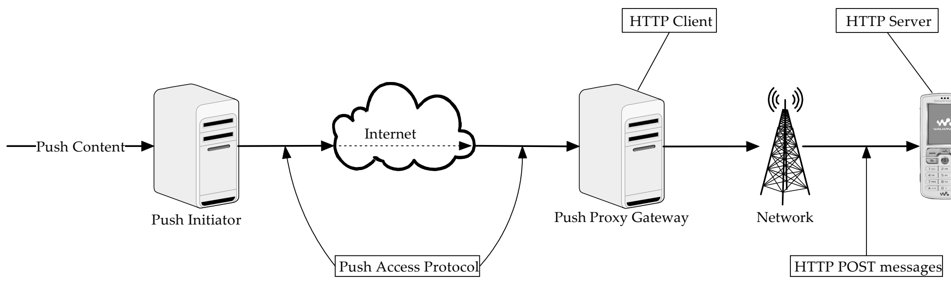

Figure 11. *WAP Push Architecture*

can choose to display it immmediately or to display it later. If the push message is of the type *service load*, the contents are automatically saved to the client, and the user settings determine if the contents should be shown immediately or cached.

We do admit that our architecture is similar to the WAP architecture — but we are bold enough to claim that it is more general. WAP has a few limitations that make it a less desirable candidate for event notification. First, WAP Push implies that we have mobile clients, preferably cell phones. This contrasts with our vision of the mobile phone as yet another Internet device, since this vision implies that the same functionality should be available at the desktop as in the mobile client. We know of no desktop application that supports WAP Push. Another reason for not using WAP Push is that its use is dictated by the network operators — to be able to send WAP Push messages, the sender must have an agreement with the operator, otherwise the push message will be blocked. This makes it difficult for service providers to offer the use of their services to customers in a flexible way. Also, the push messages are sent at the network operator's discretion — there is little insight for the service provider in the operator's agenda. But the biggest concern with WAP is the abscence of a pub/sub service. One could argue that, for example, WAP service load could be used to provide a pub/sub service. However, there is no way to determine if a certain client is online. This means that a publisher could, potentially, send large amounts of push messages that are not received by any client. It is also extremely difficult to determine the IP address of a client. Further, there is no way for a user to initiate a push session, which is required to enable the client to send messages to other clients. WAP Push was designed for one-way communication.

5.2 OMA IMPS Architecture

In April 2001, the Wireless Village initiative was formed by Ericsson, Motorola and Nokia (Ericsson et al., 2002). The purpose of the initiative was to provide an architecture for instant messaging (IM) and presence service. Wireless Village has since been incorporated into Open Mobile Alliance (OMA) and renamed to *OMA Instant Messaging and Presence Service Enabler* (Open Mobile Alliance, 2005b), or, shorter, *OMA IMPS*. OMA IMPS is not limited to mobile devices, the architecture is designed to accomodate any type of user. IMPS also defines how gateways to other instant messaging services[12] should function.

IMPS has four *Application Service Elements*:

- Presence
- Instant Messaging
- Groups
- Shared Content

The Presence Element enables user to gain access to information about other users, such as the status of the IMPS client software and the availability (online, away, disconnected, etc) of the user.

The Instant Messaging Element provides IM support. Messages can be sent to other IMPS users or users on proprietary systems. The messages are sent either to a single recipient or to a group of recipients. As IMPS could deliver any content type, OMA has speficied a mandatory content type that every IMPS implementation should support. This content type is UTF-8 encoded Unicode text. In addition OMA lists four more content types as *suggested* content types. These are *Multimedia Message*, *Enhanced Short Message*, *Business Card*, and *Calendar Entry*[13].

[12]For example, MSN Messenger and ICQ.

[13]These content types will not be defined here, they are included for the sake of completeness

The purpose of the Group Service Element is to let a number of users interact in a chat-room, as opposed to the person-to-person communication in IM. Groups can be private or public. Group messaging is done using the IM component, with the IMPS server responsible for distributing messages to all group members.

Finally, the Shared Content element provides a mechanisms by which users can share files, such as pictures, videos and documents, with other IMPS users.

5.3 IP Multimedia Subsystem

IMS (not to be confused with IMPS), the IP Multimedia Subsystem, is a standard defined by 3GPP[14]. The purpose of IMS is to provide network operators with an architecture for providing IP-based services and applications. The architecture is based on the SIP protocol. IMS will eventually provide services like user tracking, where a user's presence and availability are tracked across several different network types (mobile, fixed, broadband, ...)

The system is not limited to (new) packet switched phone systems, (old) circuit switched phone systems (e.g. "classic" land-lines) are also supported (Ericsson AB, 2004; Attal, 2005)

The IMS architecture consists of three layers:

1. *Application layer:* user applications, such as telephony and video conferencing, reside in this layer
2. *Control layer:* the control layer is responsible for managing call setup
3. *Connectivity layer:* this layer provides packet transport

There are, as shown in Figure 12 on the next page, many server types[15] in the IMS architecture. The figure is, in fact, heavily simplified. The Application Layer

[14]3rd Generation Partnership Project

[15]Actually, the IMS does not specify *servers* — it specifies *functions*. IMS vendors are free to implement the functions in the way they choose. This means that two functions may be handled

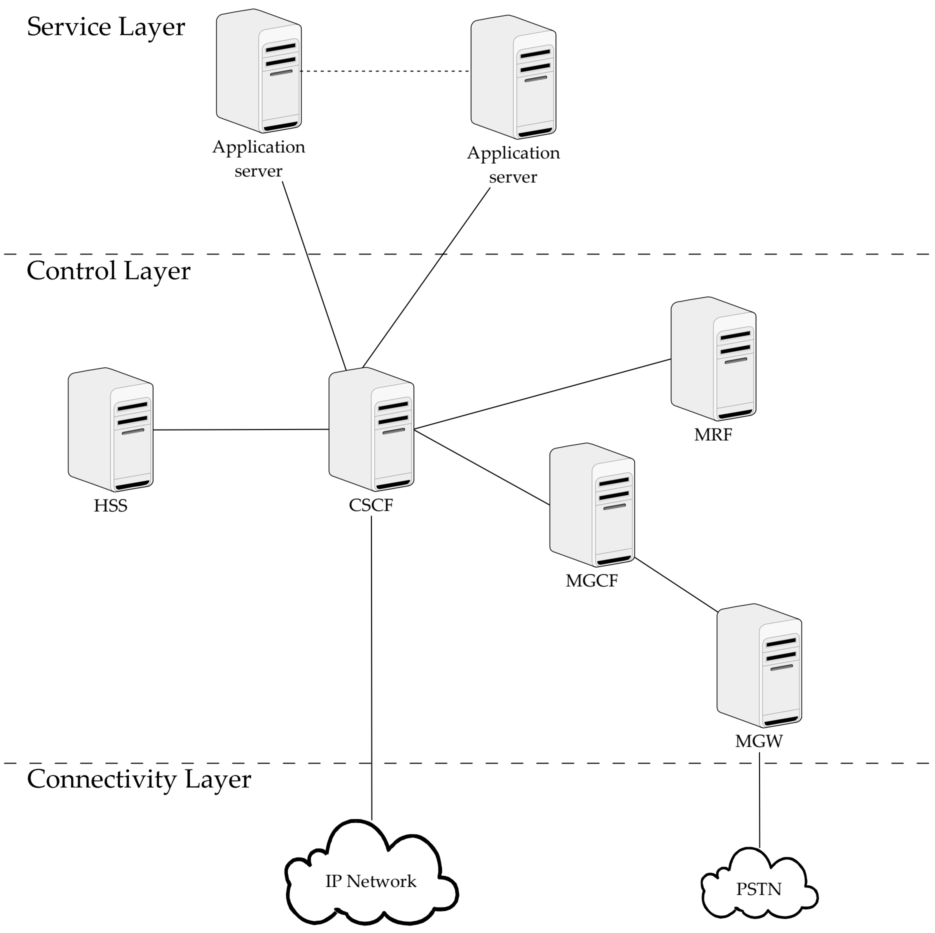

Figure 12. *IMS Overview*

is self explanatory — this layer contains different types of user applications, and is the part of IMS that is visible for end users.

5.3.1 How IMS works

Every phone is registered in a directory, the *Home Subscriber Server* (HSS). The HSS directory contains mappings from user names to IP addresses. When a user

by the same server in one vendor's IMS solution, and by two different servers in another vendor's solution. Either way, it helps to think of the different functions as distinct servers.

makes a call, the *Call Session Control Function* (CSCF), which is a SIP server, is invoked. The CSCF looks up the recipient's IP adress in the HSS and connects the two parties.

If the call is to (or from) a subscriber in the PSTN[16], the *Media Gateway Control Function* (MGCF) and *Media Gateway* (MGW) are invoked. The MGCF and MGW translates between IP based telephony and PSTN voice.

The *MRF* (Multimedia Resource Function) is used to provide media to the users. The MRF can be used to play contents, allow multimedia conferencing, and to perform data conversion, amongst others.

The above description of IMS has been heavily simplified. There is much more to IMS than what we could describe in this space. We have noted that IMS is a complex architecture, and should we have based our work on the IMS architecture we would have had very little time to concentrate on the core of this thesis, which is notification service. IMS is, in essence, an IP layer for mobile networks, to allow IP communication. Using IMS would have meant that, not only should we create our notification service, but we would also have had to develop a functioning IMS implementation. Obviously, this was not possible to do. Instead, by using a more generic architecture, we have been able to utilize existing, well documented protocols for our prototype, but the concepts are transferable to IMS. In the future, when IMS is more widely deployed, there may be no reason to use another solution. In the meantime, we believe that we have a sound solution, based on open standards and protocols.

[16]Public Switched Telephone Network

Security & User Integrity | 6

We have not paid any special attention to security issues in this thesis. The main focus has been on functionality. Nonetheless, we have identified some issues that should be addressed:

- connection security
- user authentication
- data encryption
- notification flooding

RFC 3923 specifies how end–to–end signing and encryption should be applied to XMPP. However, very few clients and servers support this at the time of writing. This RFC effectively deals with the first and second item on the above list.

XMPP uses SASL (Simple Authentication and Security Layer) for authentication.

The remaining security threat is notification flooding, whereby one or more users send excessive amounts of notifications, targeted at a single user. This could lead to denial of service for the targeted user. There is no obvious way to eliminate this threat, but there are some ways by which we believe the problem may be lessened. The first option is to limit the number of notifications that a server sends out to a user per unit of time. That approach would mean that there might still be attempted notification flooding, but the attack stops at the server. Another approach would be to have the server set thresholds for incoming notifications from a user.

When a user sends excessive amounts of notifications, the server may choose to not accept these notifications (in other words, sending many notifications has an inhibitory effect on the server), as it may be a sign of an attempted flooding attack.

Besides the technical aspects of security, there are also quite a few implications for the user of the services. In the current implementation, the user is not informed every time the application sends out data. This means that the user could be unaware that her position data is being made available on some web site. It also means that someone could activate the application on somebody's phone in order to spy on them. Even if the user *is* fully aware of the information that is sent out, there should be some way for the user to stop the data collection and subsequent publishing. An example of when information publishing could be exploited is the the service that logs a client's position. Who should have access to this information? A burglar might, for example, simply check a person's position to determine if they are at home. This problem is not solved by requiring the client to log in to a server to review position logs, since the server could be hacked. That is, however, not a situation that is unique to this example. Any server could, potentially, be compromised.

7 Prototype: A Blogging Service

A BLOG, SHORT for Web Log, is simply an online journal where readers have the possibility to comment the entries. *Blogging*, or *to blog*, refers to the activity of maintaiqning a blog. Blogging has become very popular in recent years, and many blogs have thousands of views everyday. However, the vast majority of blogs are maintained by individuals who do it for fun. Theses blogs are mostly read by friends and relatives. In a way, blogging has become the successor of personal web site. The most obvious advantage over a personal web site is that blogging requires no HTML skills — most blogs use publishing systems such as WordPress, Movable Type or blogger.com. Designer skills are of second concern in blogging, instead it is largely content driven, where the content typically is text even though audio and video blogging has become more popular over the last few years. In other words, *how* you say something does not mean so much, it is *what* you say that is in focus with blogging.

We have developed a working proof-of-concept model of our architecture. This includes a web server, an XMPP server and XMPP enabled clients. For the prototype we have concentrated on blogging features. We have tested the prototype in a real-world environment, however, the number of concurrent users was small, and so we can not say for sure what the performance would be for a large scale deployment. But since the amount of data that is transported over XMPP has a relatively small footprint, we believe that this will not be an issue. This is, however, an issue that needs to be more thoroughly investigated. Our prototype implements a simple blogging system, with both manual and automated features. The manual features include the possibility to edit and post entries to blogs, as well as reading them. The automatic feature is associated with the media player in the phone. When a user listens to a song, the ID3 information is extracted from the MP3 file and is sent to the server. The server tries to do a lookup at amazon.com to retrieve album art for the song that is playing. The album art, together with the song information, is presented on a music log. The music log is, in essence, an automated blog. Other automated features that we considered, but did not implement due to time concerns, include automatic call history publishing and SMS/MMS history publishing.

Note that the word "publishing" in this context does not necessarily imply that the information should be made available to everyone; it could also mean that data is uploaded to a private area on the Internet. For example, the following service, "Automated Call History Publishing", which we chose not to implement in our prototype due to time constraints, is a service that uploads information about a call when it is finished. The service would log the number dialed, the duration of the call, and the geographical location[17] of the start and end of the call. The location could then be put on a map to allow the user to track his movements. An SMS/MMS history service would work in a similar way. The client application of the prototype was programmed in Java, using the J2ME specification. For the music log feature, some C programming was necessary to gain access to the media player in the telephone. The server application consists of an Apache server with Struts, and a collection of servlets and Java server pages.

So, what can the prototype do? We have implemented three basic services:

- Blogging
- Photo Sharing
- Music Logging

[17]The position data could be provided by the network. However, GPS data provides higher accuracy, and is thus preferrable. With the advent of the Galileo positioning system, even higher accuracy will be achieved.
Galileo is the European Union's effort to build a satellite navigation system (see `http://europa.eu.int/comm/dgs/energy_transport/galileo/` for further information.)

The *blogging* service is a simple interface for transmitting text provided by the user. The text is sent to a server and published. *Photo Sharing* lets the user take pictures that are automatically sent to and published by a web server. *Music Logging* sends information about played songs to the web server. Photo Sharing is quite slow, due to the inherent speed limits in the cell phone networks. On average, it takes about 30 seconds to send a picture. The blogging service is less data intensive, and thus much quicker. The Music Logging service is run in the background, transparent to the user. Our implementation of event notification is very basic: the user is notified of what type of event that was received, but no other data is downloaded. One could argue that when the client receives a notification, the event should also be downloaded. This is, however, not always the best solution — there is no way of knowing at any given time that the user really wants to see the associated event. Therefore, event *notification* and event *delivery* should, in our opinion, be separated.

7.1 Client Software

Most modern mobile phones have the ability to run third party code in some form. This is not really a new feature, but even though this has been possible for some years it has never really caught on with the users. Some users download games, of course, but that is about it. One might ask why more advanced applications have not had more success? There are probably many reasons. We believe one might be that the applications that are available today have not been integrated with the rest of the phone's software; for an application to be successful the user should not even have to think about it when she uses it. The best example of such an application is the one used to dial a phone number shown in Figure **??**. The user never thinks about that she is actually using a piece of software when she dials a phone number, many users probably do not even know it. The bold goal for our software was that it should be as easy to use as the dialing application.

During the initial planning of the work we had to decide on what programming languages and technologies we would use during the development of our client software. There are currently two ways to develop software for the Sony Ericsson phones, either by using C and OPA, or by using J2ME. Both has its pros and cons. OPA has a higher entry level than J2ME, but provides much more access to the underlying system. J2ME, on the other hand, makes it easy to develop prototypes. Since both authors had much experience Java and none with OPA and this project only was intended as a research experiment with a limited time schedule we decided to go with Java as it would allow us to develop more rapidly. Since we had access to the system source code, we could develop our own Java APIs to acess those parts of the system that we needed. But developing Java APIs is time consuming and error prone. Developing APIs to integrate our software with the rest of the phones application the way we wanted was not feasible. It would be easier to go with OPA. This choice made it virtually impossible for us to fullfill our original goal, but it was a sacrifice that had to be made.

7.1.1 Architecture

One goal when designing our architecture was to make the software as modular as possible. All functionality was implemented as plugins, which means that it is easy to add new functionality to the application. There were several reasons for this approach. One was that we wanted it to be easy to extend the software with new features in the future.

The prototype uses the notion of *event generators*, *dispatchers* and *transports*. An event generator is a piece of software that injects an event into the system, where an event for example might be a blog entry or a photograph. The event dispatcher is responsible for routing this event to a suitable transporter that knows how to deal with the event. The architecture permits any number of transporters and generators, one might for instance route photographs to Flickr!, blog entrys to Blogger.com and diary entrys to diaryland.com. Choosing this design allows for a great deal of decoupling. Generators does not have to know about the different types of transporters, and the transports only need to know about how to deal with the events they care about. The J2Me specification does not currently allow an application to load code dynamically, but this might be supported in future versions. If such support was to be added, our design would easily allow for third party generators and transports (we like to call them Bloglets) to be downloaded of the Internet and run from the phone.

7.1.2 Java 2 Platform, Micro Edition

Java 2 Micro Edition (J2ME) is a framework developed by Sun Microsystems to provide Java support on embedded devices such as mobile phones, Personal Digital Assistants or any other small devices with limited computional power and memory. J2ME is not one single standard but rather a group of individual JSRs (Java Specification Requests) that an implementation may support. The two most important of these regarding this thesis are *Mobile Information Device Profile* (MIDP) and *Connected Limited Device Configuration* (CLDC).

7.1.3 Smack

Smack is an XMPP-library written in Java. It is released as Open Source under the Apache License. From the Smack website[18]:

> *Smack is an Open Source XMPP (Jabber) client library for instant messaging and presence. A pure Java library, it can be embedded into your applications to create anything from a full XMPP client to simple XMPP integrations such as sending notification messages.*

To get XMPP functionality we ported Smack from J2SE to J2Me. Our port is aptly named Smack-ME (Smack Micro Edition), as it intended to run on mobile devices. Our port was based on Smack 2.0 (the current version when this is written is 2.1) and mainly achieved by stripping Smack of (non-vital) functionality not supported

[18]`http://www.jivesoftware.org/smack/`

by J2ME and by reimplementing parts of the J2SE API's in a way suitable for mobile phones.

7.2 Server

The server system of our prototype consist of two parts, a weblog system and an XMPP-server.

7.2.1 The Weblog

There are a lot of open source weblog-systems available online, but we know of no one that is XMPP aware. Therefore we decided to develop a simple one by ourself. Our system is built on Tomcat and uses the Apache Struts and Apache Cocoon libraries for presentation. For communication with the XMPP server we utilize the Smack API.

7.2.2 The XMPP server

There are several XMPP servers available and several are Open Source[19]. The different implementations all support the same base protocols but they differ in which JEPs they support. JEP-0060 is still a work in progress and there are not that many implementations currently available. We tried two; Idavoll, which is an addon for Jabberd, and the implemenation shipped with Ejabberd. We were unable to get Idavoll to work and therefore went with Ejabberd.

[19] For the interested reader there is a thorough comparison of different XMPP servers available at `http://www.jabber.org/admin/jsc`

Results & Conclusions 8

We believe that what lacks to take the World Wide Web to the next level is a notification method. We wish to merge some of the features of instant messaging with the web. While Instant Messaging provides fast interaction, the contents of conversations is by default kept private and are often of a transient nature. No immediate way of publishing IM information on the web exists. The web, on the other hand, makes information available to the world. But it is a static system, which does not allow for simple user feedback and interaction. As we have showed in this thesis, it is indeed possible to merge these two worlds of communication, producing a new and enhanced web that handles instant messaging and the serving of web pages using the same protocols. It is our firm belief that notification is necessary if new and exciting mobile services are to emerge.

8.1 J2ME

While J2ME provides quick and easy access to advanced functions in the phones, there are some issues. First, there is no way to make a Java application an integrated part of the phone's software. By this we mean that we can not produce an application that is automatically started and runs in the background. The user will have to manually start the application every time the phone is restarted, which means that she is likely to miss some notifications, due to the client software not running.

Further, the Java UI possibilities are rather limited. There is little that the programmer can do to produce an aesthetically appealing J2ME program — unless she decides to draw the entire UI by herself. This is what we ended up doing, but it is a practice that should not need to be resorted to.

As far as performance goes, we are satisfied. Once the application has started, there is no perceived slowness compared to native programs. While we tested our application, we were impressed by the network performance of the J2ME engine, which was much more efficient than what we had thought.

8.2 OPA — Open Platform API

The most obvious drawback of OPA is that, despite the name, there is nothing *open* about it — third party developers have no access to the API or its documentation. Also, any OPA application needs to be added to the phone at the same time that all other parts of the software are added. There is no way to add an OPA program to a finished phone.

However, we *did* have full access to OPA — but we still did not use it. Why, you wonder? Simply because OPA is a very large API, and our application would have changed several OPA components. Therefore, it was not feasible for us to choose this path.

8.3 XMPP Performance

While XMPP is standardized and used in many clients, there is the issue of performance. As XMPP is based on XML, it is a verbose protocol that has a large overhead when sending data. One would imagine that this could have a negative impact on the performance of mobile clients. However, as we realized, the data load is still rather small, ranging in hundreds of bytes. This means that the client's Internet connection does not have to be very fast to be able to communicate using XMPP. We have come to the conclusion that XMPP is well suited to the domain of mobile computing. Even though it is a erbose protocol, it is not so verbose as to negatively affect a client's ability to participate in notification services. Of course, when very large amounts of notifications are sent to a client, it is possible that the client can not cope with the data flow. This is, however, not strictly XMPP related. The same thing could happen using any protocol.

Future Work | 9

We have identified some areas that needs further investigation. This includes issues such as implementation, data compression and user surveys. These are not issues that we regard as less important, they are rather issues that were not essential to develop a working prototype.

9.1 General Suggestions

A more thorough investigation of the IP Multimedia Subsystem would be interesting. Also, the prototype is very basic. To be able to use the service in a production environment, the application needs more work in both user interface design and functionality.

9.2 Re-Implementation in OPA

The services are implemented in the phone in Java, though all services could be implemented in the native application layer. Our reason for this is that the native application layer for Sony Ericsson phones was too complex for us to learn in the time we had available to do the thesis work. Instead, to maximize productivity, we decided to do all client side programming in Java. To achieve better transparency for end users, all parts of the application that run in the phone should be ported to OPA applications.

Another reason for porting to OPA is to make it easier to integrate pub/sub services with existing phone applications, such as the phone book, media player, and SMS/MMS storage. By integrating our implementation with the platform, it would also be easier to create native graphical user interfaces for pub/sub services.

9.3 Compression of XMPP Data

A problem with mobile clients is the slow data transfer speeds. While this is a problem that is likely to disappear as mobile data services are developed, it is still of interest to perform data compression on the XMPP data that is sent. This is because XML (and thus XMPP) is a verbose protocol that contains a lot of overhead. There are essentially two ways to compress the data. The first one is based on a common understanding between the server and client about how an XML message is composed. This approach is used in *Binary XML* (Open Mobile Alliance, 2005a), a proposed specification from Open Mobile Alliance. In this specification (which is specific to IMPS), every XML tag in IMPS has been associated with a two-byte token. This means that, instead of using an arbitrary long text string, we can build XML by using these tokens. This can lead to a significant decrease in bandwidth usage. The other approach is to have the client compress the entire XMPP message before it is sent, using some common compression library[20]. The server then has to decompress the XML data.

So, which method should one choose? Binary XML uses very little bandwidth for the XML tags. However, the data load is not compressed. Also, there is no way to add a new XML tag. Normally, when using XML, anyone can define any tag they want. To extend Binary XML, an OMA specification must first be produced

[20]For example, zlib (`http://www.zlib.net/`)

to ensure interoperability.

Using a compression library, on the other hand, preserves the spirit of XML: that anyone can define a new tag. There is also compression on the whole of the message (zlib achieves a compression rate of about 70%), which means that we probably achieve a better bandwidth usage than with Binary XML. However, binary data will have to be Base64 encoded (Freed and Borenstein, 1996) before being transported with XML. This encoding adds 33% to the data load. By compressing the Base64 data, the size increase that results from Base64 encoding can be nullified (Greenfield and Ng, 2005). While Base64 encoding provides a way to include binary data in the XML message, a better approach would be to send the data out of band. Instead of sending the data itself, an URL is sent. This makes it possible to send a much smaller XML message.

In summary, we firmly belief that compression algorithms should be favored over Binary XML.

9.4 User Surveys

While we have concentrated mostly on implementing publish/subscribe and actually get it to work in a satisfying way, not much effort has gone into investigating what potential users really want. We have no doubt that pub/sub services have a large array of possible applications, but exactly which those applications, we do not know. To address this issue, user surveys should be performed, to determine which areas that are the most interesting for pub/sub services.

References

Dennis Attal. IMS: Internet age telephony. *Alcatel Telecommunications Review*, (1), 2005. URL http://www.alcatel.com/atr/.

Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide–area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

Ericsson, Motorola, and Nokia. The wireless village inititative. White paper, OMA, 2002.

Ericsson AB. IMS — IP multimedia subsystem. White paper, Ericsson AB, 2004. URL http://www.ericsson.com/products/white_papers_pdf/ims_ip_multimedia_subsystem.pdf.

R. Fielding, U.C. Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol — http/1.1. RFC 2616, IETF, June 1999.

N. Freed and N. Borenstein. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. RFC 2045, IETF, November 1996.

Paul Greenfield and Alex Ng. A study of the impact of compression and binary encoding on soap performance. In *Proceedings of The Sixth Australasian Workshop on Software and System Architectures*, pages 46–56, Brisbane, Australia, 2005. Swinburne University of Technology.

Joe Hildebrand and Peter Saint-Andre. JEP–0080: User geolocation. JEP–0080, Jabber Software Foundation, May 2005.

René Meier and Vinny Cahill. Taxonomy of distributed event–based programming systems. *The Computer Journal*, 48(5):602–626, 2005.

Peter Millard, Peter Saint-Andre, and Ralph Meijer. JEP–0060: Publish–subscribe. JEP–0060, Jabber Software Foundation, March 2005.

Open Mobile Alliance. Client–server protocol binary xml definition and examples. Specification, OMA, 2005a.

Open Mobile Alliance. IMPS architecture. Architecture document, OMA, 2005b.

Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Core. RFC 3920, IETF, October 2004a.

Peter Saint-Andre. Extensible messaging and presence protocol (xmpp): Instant messaging and presence. RFC 3921, IETF, October 2004b.

Peter Saint-Andre. Mapping the extensible messaging and presence protocol (xmpp) to common presence and instant messaging (cpim). RFC 3922, IETF, October 2004c.

Peter Saint-Andre. End–to–end signing and object encryption for the extensible messaging and presence protocol (xmpp). RFC 3923, IETF, October 2004d.

Peter Saint-Andre. Streaming xml with jabber/xmpp. *IEEE Internet Computing*, 9(5):82–89, September 2005.

Bill Segall and David Arnold. Elvin has left the building: A publish/subscribe notification service with quenching, 1997. URL `http://elvin.dstc.com/doc/papers/auug97/AUUG97.html`.

Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with Elvin4. In *Proceedings AUUG2K*, Canberra, Australia, June 2000.

Glossary

3GPP	3rd Generation Partnership Project
API	Application Programming Interface
CGI	Common Gateway Interface
GALILEO	The European Union's initiative to build a satellite positioning system
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID3	A metadata format for MP3 files that adds information such as song title, artist, album, etc. to the MP3 file
IETF	Internet Engineering Task Force
IM	Instant Messaging
IMPS	Instant Messaging and Presence Service
IMS	IP Multimedia Subsystem
J2ME	Java 2 Micro Edition
JEP	Jabber Enhancement Proposal
JSR	Java Specification Request

MMS	Multimedia Message
MP3	MPEG-1 Audio Layer 3, a lossy format for audio encoding
MPEG	Motion Picture Experts Group
OMA	Open Mobile Alliance
OPA	Open Platform API
REST	Representational State Transfer
RPC	Remote Procedure Call
SASL	Simple Authentication and Security Layer
SIP	Session Initiation Protocol
Smack	A Java API for creating XMPP clients
SMS	Short Message Service
SOAP	an XML-based protocol for exchanging structured and type information
WAP	Wireless Application Protocol
WSP	Wireless Session Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Index

3GPP, 32

Binary XML, 48

Call Session Control Function, 33
CSCF, 33

Extensible Markup Language, 23

iChat, 26
ID3, 38
IMPS, 31
IMS, 32
IP Multimedia Subsystem, 32

Jabber, 23

M2M, 27
Machine to Machine, 27
MP3, 38

OMA, 31
OPA, 4
Open Mobile Alliance, 31
Open Platform API, 4

Publish/Subscribe Systems, 11

SIENA, 15
SIP, 32
Smack, 41
Sony Ericsson, 4
Sun Microsystems, 41

XML, 23
XMPP, 23, 41

zlib, 48