TABLE OF CONTENTS

CHAPTER 1

Introduction

The area of Presence is quite a huge one with many specifications and drafts specified by the IETF and then the Open Mobile Alliance (OMA) has taken those various drafts and specifications and brought them into one unified context. OMA has also extended some of the specifications as well as clarified the existing ones when there have been ambiguities. The purpose of this book is to go through all the relevant specifications and drafts made by the various standardizations bodies and during the course build up a Presence System from ground up. At the end of this book the reader should have a deep knowledge of all the relevant parts in the vast area of presence.

1.1 Targeted Audience

This book is written by a professional for professionals. It is assumed that the reader is familiar with general networking concepts, with IETF

1.2 The Layout of This Book

Every chapter will start with a problem description and what can be done to solve this problem and then continues to describe the problem and solution at hand in an introduction way. Any new concepts and definitions are discussed and defined within the introduction and once the general language has been set it continues to look into the problem in more detail. However, the main discussion does not dive into every little detail but rather tries to highlight the common use cases. For architects, designers and developers that really need to know all the details, each chapter has a detailed section about the problem and solutions at hand. At the end of each chapter you will find a Summery section that yet again states the problem and the solution found in the chapter. It also makes the transition to the next chapter.

1.3 What is Presence

Talk about presence, what it is and how it is commonly used. Extended usage of presence information includes to use it when making routing decisions etc.

Let us just look at the very basic user scenario and also the most widely used.

Introduce Alice and Bob, have Alice subscribe to Bob, show how this information is conveyed to Alice.

Presence is not just one specification but rather quite many specifications out of IETF and then OMA has taken all of those and put them into a context in order to form a full blown Presence System. The underlying technology is SIP so therefore it is important that someone who wants to know all the details of Presence also is familiar with the SIP protocol. Therefore, before we get into the details of Presence we will first go through the basics of the SIP protocol.

1.4 The Open Mobile Alliance

Who they are, what they do, why they are important for us in this context…

CHAPTER 2

SIP

Presence is based off of SIP so therefore an introduction to this area seems to be appropriate. However, SIP itself is a huge topic and since this book is concentrating on Presence blah blah blah…

2.1 History of SIP

Blah

2.2 HTTP V.S. SIP

Blah

2.3 Base SIP

2.3.1 Request and Responses

2.3.2 The METHOD

2.4 SIP Dialog

The most common usage today of SIP is to use SIP for setting up and maintaining "phone calls" (VoIP).

However, it is important to realize that a dialog is not bound to the SIP INVITE message but rather is just a general concept within SIP and describes how a relation between two SIP endpoints can be described and maintained. In RFC 3261, the only SIP method defined to setup such a relationship is the SIP INVITE message but as we will see later one, there are other methods that is capable of establishing a peer-to-peer relationship between two end points.

2.5 Summary

In this chapter we went through the basics of SIP from a Presence perspective. As pointed out, there are much more to SIP and they are many books on the subject but the best source of truth is really just to read RFC 3261, it contains everything you will ever need to know about SIP.

Now that we have the basic understanding of SIP we can continue to examine the RFC that lays the very foundation of a presence framework, namely RFC 3265 who discusses a general SIP based event notification framework.

CHAPTER 3

Event Notification Framework

Applicaitons often face the problem of registering an interest in the state of a particular resource and then getting notifications when that resource changes. A resource can really be anything, it could be a particular document on your local hard drive, or why not a document on some server far away. It could be the state of your voice mail, the state of the CPU or why not the presence state of your best buddy? Or to push the discussion even further, why not the state of the particular bus you take to work? The point is that all of these things could be considered as a Resource and you may want to know what those Resources are up to. Since this is a common problem, IETF set out to specify a general Event Notification Framework that used SIP for allowing end points to register an interest on a particular Resource and then be notified when that Resource changed state.

This chapter will go through everything you need to know about this SIP based event notification framework and will be the ground for everything else in this book.

3.1 Introduction

On a high-level, Figure 3-1 illustrates what we are trying to accomplish here. We want to setup a relationship between our end-point and a Notifier (who has knowledge about the Resource, which is the one we are really interested of) so that whenever the state of that Resource change, we will find out. Of course, this relationship will last for some time and the way to do that in SIP is to establish a dialog. In the chapter about SIP we saw that in the base line SIP specification, the only way to establish a dialog (and therefore also a lasting relationship between two endpoints) was to send out an INVITE request. However, we also pointed out that a dialog is a general concept and that there are other methods that are capable of establishing a SIP dialog. RFC 3265 introduces the SIP SUBSRIBE message and clients use it when they wish to register an interest in the state of a Resource.

The state of the Resource is sent out by the Notifier to the Subscriber through a NOTIFY request. This request contains the actual state of the Resource as well as information about the state of the Subscription itself. The NOTIFY request is also a new SIP method

introduced by RFC 3265 and before we get into any more details let us first establish the general concepts of introduced by RFC 3265.

[insert picture here]

Figure 3-1 shows how Alice asks to get updates of the state of her voice mail.

3.1.1 A General Framework

RFC 3265 is the IETF specification that lays out the fundamental framework for a SIP based Event Notification system, which allows for SIP UA:s to register an interest in a particular Resource and then to be notified when the state of that Resource is changed. However, it is important to realize that this framework is a general framework and has actually nothing to do with Presence. This is a common misconception but an understandable one since, at the time of writing, presence is probably the most widely used and most well known event package deployed onto this framework.

We have already started to use terms such as a “Resource”, “Notifier”, “Subscriber” etc but so far we have not really defined what those really are. So, before we go any further, let us first establish a common language by examining the concepts introduced by RFC 3265.

3.1.1.1 The Subscriber

The Subscriber is the SIP end point that registers an interest for receiving state changes of a particular Resource. The Subscriber accomplishes this by issuing a SIP SUBSCRIBE request, which is processed by a Notifier. If the request is accepted, the Subscriber will have installed a Subscription associated with that Resource and will be receiving notifications of any state changes in that Resource as long as this Subscription is alive.

3.1.1.2 The Subscription

The Subscription represents a peer-to-peer relationship between two SIP end-points, the Subscriber, and the Resource (or rather the Notifier, who has knowledge of the Resource and its associated state). Looking at the SIP level, a Subscription is represented by a SIP dialog and as long as this dialog is valid, the Subscription is valid, and as long as the Subscription is alive and well, the Subscriber will be receiving state changes.

Might remove this comment and rather mention it later on along with this discussion in the chapter about privacy rules – Note that there might be other factors that actually prevent the Subscriber of receiving the true state of the Resource, such as privacy filters.

Compare the Subscription with any real world subscription to e.g. your local newspaper. You as the Subscriber will create a subscription with the news agency and upon approval your subscription will be installed and activated. As long as you want to and as long as you pay your bills on time, your subscription will be active and you will be receiving a copy of the newspaper every morning.

3.1.1.3 The Notifier

The purpose of the Notifier* is to processing incoming subscribe requests and to send out notifications to subscribers in order to convey the state of a particular Resource. If a subscription is accepted, the subscription will be installed and that subscriber will consequently be receiving notifications for as long as that subscription is active. One of the more important parts of processing a subscription request is to decide whether to allow the request or not. Since RFC 3265 just defines a general framework, the actual decision is left up to the implementing event package and depending on that event package; the authorization decision may be a very simple one or quite complex, as in the case of the Presence Event Package (which we will go through later on)

3.1.1.4 The Resource

The most central component of this entire framework is the Resource (and yet it is not explicitly defined in the specification). It is to a particular Resource a SIP end-point registers its interest in and it is the state of this Resource that gets conveyed to the Subscriber by notifications sent out by the Notifier. As mentioned in the introduction to this chapter, a Resource can really be anything and what a Resource really represents it is up to the implementing Event Package.

The central concept of this entire framework is the resource.

REMEMBER! The Resource is actually just the request-uri but the way I define it here is request-uri + event package. Or even more,

3.1.1.5 The Event Package

An Event Package is other specifications that describe what a Resource really represents. It is up to the implementing Event Package to define what state is conveyed to the Subscriber and also the format of that information along with any additional rules required to parse that state. Another important part of an Event Package is to define how authorizations of Subscribers are to take place, i.e. is Alice allowed to subscribe to the state of Bob?

3.1.1.6 Template Event-Package

A template event-package is a special kind of event package that cannot operate in isolation but needs to be applied on top of another event package.

3.1.1.7 The Subscribe Request

It is through constructing a sending a SUBSCRIBE request a SIP end point can establish a subscription with the Notifier. The SUBSCRIBE is a dialog creating SIP method so therefore, upon a successful request, a dialog will have been established on both sides and that dialog will represent the relationship between the two end-points.

3.1.1.8 The Notify Request

Once a subscription has been installed, any changes in the state of the Resource is sent out using a NOTIFY. This is also a new SIP method introduced by RFC 3265 and is always sent within the dialog created by the Subscribe request* (even though there are cases out in the real world where implementers actually are sending out un-solicitated NOTIFY requests. Not very good and it is breaking the spec – TODO, check up what the spec really says. Also check up this that I only think Microsoft is doing).

The most important part of this request is to carry the state of the Resource and as previously mentioned it is up to the implementing event package to define exactly how this information is structured.

Another important part of the Notify request is to carry information about the actual state of the subscription. This is conveyed through a new SIP header, namely the Subscription-State header. In this header you will find information if the subscription is active, when it is about to expire, if it is terminated and what the reason for that was etc. We will see examples of how to interpret this header later on and also examples of how it is used.

3.2 Establishing a Subscription

In Figure 3-1 we saw at a very high-level of where a Subscriber, Alice, registered an interest in the state of her voice mail. Even though that that flow did not go into any SIP specific details we have already hinted that the Subscriber needs to construct and issue a SUBSCRIBE request and then send that over to the Notifier. Upon a successful reply, the Subscriber will be receiving notifications about any changes in the state of the Resource until the subscription is terminated. So, let us now go into more details about the initial SUBSCRIBE request.

Of course, a first important step is to indicate which Resource you are subscribing to and the request-uri contains exactly that. E.g., if someone wanted to know the state of Alice's voice mail that someone would need to insert the sip-address of Alice into the request-uri – sip:alice@example.com. However, this address do not convey if the subscription is for the state of her voice mail box, the state of the inbox of her email or perhaps her presence state. Which information you are interested in is conveyed in a new SIP header, the Event-header. The value of this header is identifying the Event Package (3.1.1.5) to which the subscribe request is targeting. This header plus the request-uri together uniquely identifies both the Resource to which you are subscribing to but also what information belonging to that Resource you are interested in.

TODO – now I am stating what 3265 states but I normally say that request-uri + event-header = Resource. Need to make sure that I am consistent in my own thoughts here. Perhaps I need to re-write this to define it according to my view… very important!

```
SUBSCRIBE sip:alice@example.com SIP/2.0
To: sip:alice@example.com
From: sip:alice@example.com;tag=123
Event: voice-mail
CallId: 987
```

Listing 3-1 SUBSCRIBE request requesting information the state of Alice's voice mail box

Figure 3-2 shows the typical call flow of someone establishing a subscription to a particular Resource. In this particular example it is Alice that is subscribing to the state of her own voice-mail box as the above SUBSCRIBE request outlines. Hence, Alice constructs the request and sent that out, which is terminated by the Notifier. The Notifier will examine the request and determine if it is capable of handling the indicated event-package and if so it will send out a 2xx response. In the flow we see that the Notifier do accept her request with a 202 Success response and then shortly after the Notifier sends out a NOTIFY request that Alice responds to with a 200 OK.

[TODO – insert call flow]

Figure 3-2 illustrates the call flow of Alice establishing a subscription towards her own voice-mail box

One thing that is very important to remember is that if the subscriber receives a 2xx response to the subscribe request then it is guaranteed that it will also receive a NOTIFY request, always! Hence, if the Notifier do accept a subscribe request then it MUST also send out a NOTIFY within reasonable time (and of course, a "reasonable" time can really be anything and is not mandated by the specification). This behavior will prove very useful later on but also introduces some problems in real life scenarios, as we will discuss in section 3.7.

3.2.1 The 202 Accepted Response

In Figure 3-2 we saw how Alice got back a 202 Accepted as a response to her subscribe request and this is a new response code introduced by RFC 3265. It merely means that the subscribe request was understood and accepted but it doesn't mean that Alice is or isn't authorized to actually see the true state of whatever it is she subscribed to.

[TODO – do a link to the discussion in the presence event package where it becomes more obvious why we need this response code]

3.2.2 The Notify request

Apart from carrying the actual state of the Resource in the body, the notify request also contains useful information about the actual subscription itself. RFC 3265 introduces yet another SIP header, namely the Subscription-State header. This header conveys the state of the subscription, such as "active", "pending", "terminated" and potentially the reasons of why the subscription ended up in that particular state. Unless the subscription is terminated, the Subscription-State header will also contain how much longer this subscription will be kept alive. This is communicated through the expires-parameter.

```
Subscription-State: active;expires=3599
Subscription-State: pending;expires=600
Subscription-State: terminated;reason=timeout
```

Listing 3-2 shows three different examples of a Subscription-State header

Examining the notify request that Alice got back in the flow as shown in Figure 3-2 we can see that her subscription is active and will live on for another 3599 seconds.

```
NOTIFY sip:192.168.0.100 SIP/2.0
To: sip:alice@example.com;tag=555
From: sip:alice@example.com;tag=123
Event: voice-mail
CallId: 987
Subscription-State: active;expires=3599
```

Listing 3-3 outlines the NOTIFY request coming back as the result of the successful installation of the

3.3 Keeping the Subscription alive

When a request creates a state it is common to have some kind of timer associated with that state so that if the state is not explicitly removed, it will be deleted when the timer fires. The reason for this is of course that clients can crash, networks can go down etc and we do not want to keep state longer than necessary. This is also true for the subscription installed by the Subscriber. A subscription has a life span and as long as the subscription is considered to be alive the subscriber will be receiving state updates. As a consequence, the subscriber must make sure that the subscription doesn't time out and therefore also terminated. So, when the subscriber constructs the subscribe request she can indicate the desired life-span of the subscription by putting in a time in seconds into the Expire-header. If the server accepts that time, the subscription will live on for that amount of time. Of course, how do we know if the server (Notifier) actually accepted the time that we suggested? The 2xx response will convey what time we actually got and the NOTIFY will actually also contain the time left.

If Alice added e.g. Expire=3600 into the subscribe request outlined in Listing 3-1 and that time was accepted by the Notifier, the 202 Success response in the call flow shown in Figure 3-2 would contain the following:

```
202 Success
show the expire header thingie
```

Listing 3-4 a closer look at the 202 Success response

Note that the Subscriber do not have to put in a suggested time (by using the Expire-header) and if the Subscriber doesn't do this then it will be up to the Notifier to choose whatever it likes. However, if the subscription is accepted, the 2xx response will always contain the expiration time and it is this time the Subscriber should set its internal timer to.

So, now we know everything there is not know about the initial timeout on the subscription but how do we refresh it so it doesn't terminate? Well, it is very straight forward. Just create a sub-sequent SUBSCRIBE request and send that one over to the server before the timeout occurs. This will reset the timer on the server side and you will be getting a new 2xx response with a new expiration time in it (most likely the same value as in the first 2xx response). A typical so-called re-subscribe request will look something like this:

```
SUBSCRIBE sip:alice@example.com SIP/2.0
To: sip:alice@example.com;tag=555
From: sip:alice@example.com;tag=123
Event: voice-mail
CallId: 987
Expire: 3600
```

Listing 3-5 a typical re-subscribe request

As Listing 3-5 illustrates, the re-subscribe request is really no difference than the initial one except that it will of course be sent within the context of the established dialog (wouldn't be a sub-sequence request otherwise, now would it!). This request will cause the server to reset the timer associated with this subscription and will give, in this example, Alice's subscription another hour to live. Note though that the 2xx response will still contain the true expiration time that the server sets so the Subscriber must always take this value and adjust its internal timer to match it. Hence, do not ever trust whatever you put in as the Expire-value and use that to set your timer on the client side.

Alice will be sending out these refresh subscribe requests as long as she wants the subscription to be kept alive but eventually she will want to terminate the subscription. In a typical use case, this would happen when Alice shuts down her client, such as when she turns off her computer to go to bed at night. So the question is then, how do we terminate the subscription?

Also, what would happen if the subscription does timeout? I.e., we didn't refresh the subscription in time. The Notifier is actually responsible for telling the subscriber when a subscription has been terminated and why it was terminated. We will discuss this also in the next section, section 3.4, and will also look at those different reasons.

3.4 Terminating the Subscription

Any state that a client have installed on remote UAS:s it should clean up when it goes offline and this is also true for any subscriptions that the Subscriber did install. We have seen that if the client is ill-behaved, there are timers to prevent this state from living on forever but of course we should really clean up after ourselves, hence, we need a way of terminating subscriptions.

Terminating subscriptions turns out to be as simply as installing or refreshing them. Simply issue a sub-sequent subscribe request where the Expire-header is set to zero. I.e., you are asking the subscription to live for another zero seconds; hence, you are

terminating it. If successful, you will be receiving a 2xx response to the subscribe request and one last notify request. The reason why you get a notify request once again is aligned with what we have previously stated, which is that for every successful subscribe request you will always be getting a notify request.

```
SUBSCRIBE sip:alice@example.com SIP/2.0
To: sip:alice@example.com;tag=555
From: sip:alice@example.com;tag=123
Event: voice-mail
CallId: 987
Expire: 0
```

Listing 3-6 an example of a un-subscribe request

If the client didn't issue a un-subscribe request but all of a sudden do receive a terminating NOTIFY with the reason stating "timeout" then the subscriber forgot to refresh the subscription in a timely manner. The Subscriber can then simply just create a new subscription if it likes and as such, just start over and pretend like nothing ever happened. A well behaved client will do this without the end-user even noticing this. There are many situations where a well-behaved end client following RFC 3265 will automatically "fix" the subscription without even telling the end-user that it did so. Some of those situations are discussed the detailed section below.

```
NOTIFY sip:192.168.0.100 SIP/2.0
To: sip:alice@example.com;tag=555
From: sip:alice@example.com;tag=123
Event: voice-mail
CallId: 987
Subscription-State: terminated;reason=timeout
```

Listing 3-7 shows the terminating notify

3.5 Polling

Polling is a technique commonly used in systems where it is not possible to register interest for state changes and to receive those changes asynchronously, as what we have been discussing in this chapter. However, even with RFC 3265 in place there may be cases where one would like to do a poll [TODO – come up with some good example, if not re-write this sentence]. Due to the fact that every successful subscribe request always will be receiving a notify we can simply create an initial subscribe request but set the Expire header to zero right away, as the listing below illustrates.

```
SUBSCRIBE sip:alice@example.com SIP/2.0
To: sip:alice@example.com
From: sip:alice@example.com;tag=123
Event: voice-mail
CallId: 987
Expires: 0
```

Listing 3-8 A poll request

This will cause the subscription to be installed and deleted right away and if this request is accepted by the Notifier it must send out that notify to us. Of course, the notify will have a Subscription-State header stating that the subscription has been terminated and the reason would be “timeout”, which in affect will be a poll request.

3.6 Subscription Life-cycle

We have already seen that a subscription can be in different state and the state in which the subscription is in is conveyed through the Subscription-State header in the NOTIFY request. So far we have really only briefly mentioned the two states “active” and “terminated” but there are more states. Figure 3-3 shows the state diagram over the various states a subscription can take on. It also shows the events that moves the subscription from one state to another. These events are quite important and will be discussed throughout this book.

[TODO – insert subscription state diagram here]

Figure 3-3 Subscription state diagram (adapted from RFC 3857)

When a subscription arrives at the Notifier the subscription enters the init state. This is just a transient state and the next state depends if the Notifier has a policy already at hand or not. The policy dictates if the subscriber, e.g. Alice, is allowed to subscribe to this resource or not. If the policy says that she is not, the subscription is moved over to the terminated state immediately and as such the subscription is denied and a 403 response will be sent out. Remember, since the subscription was not accepted the subscriber will NOT be receiving a NOTIFY request. This is aligned with what we previously have been discussed.

If the policy states that the subscriber is authorized then the subscription moves over to the “active” state and the subscriber will be receiving a 2xx response followed by a NOTIFY within a reasonable amount of time. Once a subscription has entered the “active” state, the only way the subscription can go is to the “terminated” state. There are several reasons of why the subscription goes from the “active” to the “terminated” state and the most common one is that the subscriber simply terminates the subscription through a un-subscribe request. In this case, the event that took the subscription from the “active” to the “terminated” state would be “timeout”.

A common case is also that there is a policy available to the Notifier but that policy does not state what to do with e.g. Alice’s subscribe request. In this situation the subscription will enter the pending state and while in this state the subscriber will only get “false” notifies back. What I mean by false here is that the state carried in the body of the notify will not necessarily carry the correct state information of the resource the subscriber subscribed to. Remember, according to the specification the subscriber MUST receive a NOTIFY for every successful subscribe request and since the subscribe request was successful the subscriber must be getting that notify. However, since the information carried in the body of the notify can be sensitive there must exist a policy that tells the

Notifier what information it is allowed to send out to that particular subscriber but as long as it doesn't have that it will send out notify requests with a body revealing nothing, hence a "false" notify.

[TODO – more info about the states?]

3.7 Details

The devil is always in the details and there are many issues that have been left out in the previous discussion in this chapter. This has been done on purpose since too many details early on may not be of any use. However, if you feel comfortable with the general behaviors described so far then continue reading!

3.7.1 State deltas

The body of the NOTIFY can really contain anything and as such, there might be event-packages that allows the body of the NOTIFY to only contain the partial state. This will of course mean less bandwidth used and may be extra important for clients operating in low-bandwidth networks such as in a mobile network.

If an event-package do allow for deltas to be sent out they must adhere to the following rules mandated by RFC 3265:

- Any NOTIFY being sent out as a direct result of a SUBSCRIBE request must contain the full state of the Resource. This behavior allows client to fetch the full state of the Resource in case they got out-of-synch
- Notifications sent out due to change in state will only contain the actual state change. In order for Subscribers to keep track of the state it receives there must be a version number as part of the state information and this number must increase by exactly one for each notification.

3.7.2 Subscription expiration

We have seen that the Subscription will be associated with a timer that, if it fires, will delete the state associated with this Subscription. The Subscriber may suggest an expiration timer in the subscribe request but it is still up to the Notifier to pick and choose (almost) whatever it likes.

3.7.2.1 A million years?

So, can you as the Subscriber suggest any time you like? Yes, you may suggest any time you like but what will happen if you do suggest a million years is that the Notifier will most likely choose another time. The Notifier is likely to be configured with a maximum allowed expiration time and if someone either suggests a time bigger than this one, or no time at all, it will pick this configured time as the expiration time. Remember, the

subscriber should always examine the 2xx response since this is where the timeout chosen by the Notifier will be conveyed.

The philosophy here is that if you have too big of a timeout, the likelihood of something going wrong is too great. I.e., the Subscriber may crash and therefore not be able to delete Subscription and as such, the state over at the Notifier may live on, consuming resources, almost forever. At the same time, you do not want to have too short of a lifespan, since this will cause more refresh traffic to flow through the network. Hence, it is likely that the Notifier will pick its biggest configured timeout.

3.7.2.2 One second?

Of course, you could suggest a expiration time of one second but you will probably get back a 423 Interval Too Brief response back. Why? Well, too short expiration time will lead to lots of refresh traffic, as we previously mentioned, and too much traffic may take down the network. Therefore, the Notifier will also most likely be configured with a minimum allowed expiration time and if you suggest something less than then, it will reject your subscription request the 423 response. This response will also contain the minimum allowed expiration value and is conveyed through the "Min-Expires" header.

[TODO – find out why we do a 423 here and not lengthen it. Think it is stated in 3261]

3.7.3 Subscription/dialog management

The SUBSCRIBE request is a dialog creating request and as such there will be points in time where this dialog is created and then eventually destroyed. The dialog and the subscription are really the same thing so when the dialog is destroyed, so is the subscription and vice versa. At first it may be obvious what those points in time are when the dialog is created/destroyed but there are common misunderstandings when this actually happens. Some of those misunderstandings leads to SIP elements (both UA:s and proxies) breaking the subscribe-notify chain, which in the worst case also will break the functionality of the service we are trying to give out (such as presence).

3.7.3.1 Dialog establishment

The subscribe dialog is of course established on a 2xx response to an initial SUBSCRIBE request, nothing strange about that. However, what UAC's must be aware of is that the NOTIFY may actually arrive before the 2xx response and if it does, then it is the NOTIFY request that establishes the dialog. Hence, answering something along the lines of 489 to that NOTIFY is wrong. This scenario could of course happen when we are sending SIP over an unreliable network and the notification simply overtakes the 2xx response on their way to the UAC.

3.7.3.2 Tearing down the dialog

When the dialog is established is fairly straight forward but when the dialog is torn down is, in my experience, a common source of error.

Without giving it much though, you may be inclined to say that the dialog as destroyed when the transaction of the un-subscribe terminates. Well, this couldn't be more wrong. First of all you of course need to at least see if you get a successful response to that un-subscribe request but even so, the dialog (and the subscription) lives on for as long as the Notifier says it does. In fact, you may think of the un-subscribe request merely as a suggestion to the Notifier that it can kill off the subscription and it is not until the Notifier agrees with you that the dialog is torn down.

So, when does this happen? The dialog/subscription can only be considered as dead when you see a final response to a terminating NOTIFY. Why this terminating notification was sent out is of no interest to us in this discussion. It may have been because of a un-subscribe request, the subscription timed out, local policy said to kill it off there are simply many reasons to why a subscription is terminated.

So no matter where in the chain of this dialog you are, you cannot clean up the state associated with this dialog until you see that final response to the terminating dialog. Since the only way to detect if the notify is indeed is a terminating notification, everyone in the chain need to examine the SubscriptionState header. This goes for stateful proxies too!

But what go so wrong if the elements in the network do not do this properly? Well, let's say that the Subscriber (the UAC) gets this wrong and terminates the dialog as soon as it has gotten the 2xx response to her un-subscribe request. We know that the Notifier (the UAS) will send out a last NOTIFY because there will always be a notification sent out to every successful subscribe request, which will cause the UAS to respond with a 489. Of course, a well-written Notifier really will not care that much since any final response to the terminating notify will cause it do clean up all its state anyway. However, if the client isn't there anymore and this is sent of UDP the Notifier will sit and wait until the transaction times out (64 * T1, which default is 32 seconds), causing un-necessary resources to be consumed.

A Subscriber who gets this wrong is not as bad as a proxy screwing things up. The functionality of the Subscriber may rely on the fact that it will get this last terminating NOTIFY so if a SIP proxy kills off its state as soon as it sees the 2xx response to the un-subscribe request, it will mean that when the NOTIFY is sent out by the notifier, the proxy will reply 489 and as such, the Subscriber will not even get the notify request!

This last case is illustrated in the call flow shown in

[TODO – insert call-flow here!]

Figure 3-4 shows how a bad proxy breaks the subscribe-notify chain

3.7.3.3 Timers in the network

The subscriptions themselves have a well-defined life span, which we have discussed quite extensively by now. However, there might be other timers in the network that, if not

maintained properly, will fire at the wrong time and as such break the event chain once again.

The Java SIP Servlet Specification (JSR116) is becoming more and more popular as a way of creating various types of SIP Applications. In this specification you will always end up creating a state (at the top-level, a SipApplicationSession) whenever you create or receive a request and it is up to you as the programmer to clean up this state after yourself. However, if the developer does not do this there is a timer associated with this state that if it fires the JSR116 container may remove this state. If this state is removed the underlying dialog (if there was one to begin with) will also be removed causing any sub-sequent requests floating through this element be terminated with a 489 response.

So, if you have a stateful application sitting on top of JSR116 that is in the path of our subscribe requests it must not forget to deal with this timer appropriately as well. Even if you have thought about the other problems described above you must not forget to configure this servlet timer appropriately as well. The default value of the servlet timer is configured in the sip.xml and may be easy overlooked. If you have built an application dealing with subscribe requests you should examine the 2xx response to find out the expiration time of the subscription and adjust the 116 servlet timer accordingly. Note, it may refuse your request so then you need to come up with another strategy in order to prevent the servlet timer to kill off the state and as such, breaking the chain causing sub-sequent requests to fail.

- NOTIFY arriving before the 2xx response and is therefore establishing the dialog. Here is something that e.g. JSR116 does not specify, i.e. can an application send out a NOTIFY before it generates the 2xx response?

- The various states of the subscription. I.e., init, pending, active, terminated and how it goes from one state to another… mention the correlation to the Subscription-State header and point out once again that this is the header that conveys this stuff…

- Keep adjusting your timer based on the stuff in the NOTIFY – stupid and just makes it complicated. Just keep it simple…
- When to re-subscribe? Just before the timer expires? Any other strategy?
-
-

- Successful subscribe request always yielding a NOTIFY can be problematic in real life scenarios. Imagine the following case: an end client is being turned off and as such it will start sending out un-subscribe requests to everything it has subscribed to (typically the presence state of your buddy list). Since we know that a successful subscribe request (un-subscribe also being a subscribe request but with expires=0) will yield a NOTIFY being sent back to which we really should reply back with a 200 OK response the client must sit and wait for all of the NOTIFY:s coming back before actually completely shutting down. If it doesn't do that, the server side will most likely get a timeout on the request and will fall into error handling mode and if implemented correctly it will still remove the subscription but not because of a successful normal case but rather for a lack of response. Also, when you think about it, why would an end client be interested in the NOTIFY coming back? Today, the most widely use of RFC 3265 is presence and most often that information is conveyed to an end-user through some kind of GUI. Hence, if the user is shutting the client down, why would she be interested in that last NOTIFY? After all, that NOTIFY should not really contain any new information about the state of the Resource she has subscribed to since she would already have received that information (unless in a corner case of course).

  Even worse, there are clients that just send out the subscribe request and then even terminates the

3.8 Summary

Now that we know how we can utilize SIP to setup a subscription towards a particular Resource and how our end point will be receiving NOTIFYs whenever the state of that resource changes we face the next problem, how do we update the state of the Resource?

Imagine the case of presence where Bob would like to convey his current state in some way. Bob might be heading off to a meeting and would therefore like to set a note that all the subscribers can view saying that he is away and busy right now. How can Bob update his presence state? I.e., how can he update the Resource representing him?

[TODO – need to re-write this to include the voice-mail case. Perhaps there is a better "event-package" to use as an example?]

We saw how Alice constructed a SIP SUBSCRIBE request to setup a subscription towards the state of her voice-mail box but we haven't Bob in order to see his presence status. However, what we haven't figured out yet is that Resource is updated. How can Bob change the state of his presence? This problem is further explored in the next chapter.

CHAPTER 4

Updating the state of a Resource

Updating the state of a particular Resource can of course be done in a number of ways. It could be done in a proprietary way, decided by the implementer of the system where external updates either is not allowed or perhaps the system exposes some kind of API allowing outside interaction.

In the previous chapter we discussed how SIP can be used to create an event-notification mechanism and we will continue building on those ideas and discuss how SIP once again can be utilized to update the state of the resource as well.

4.1 Introduction

In the below figure, Alice subscribes for the presence state of Bob. The flow shows how Alice initiates the subscription and receives a NOTIFY request containing the current state of Bob, which is offline, and then how Bob comes online and changes his state to online which Alice finds out through the last NOTIFY.

However, the question is how does Bob change his presence state? How do we update the state of a Resource, whatever that Resource may be?

RFC 3903, SIP extension for Event State Publication, does address this very problem and is as an extension event-notification framework discussed in the previous chapter. This RFC introduces a new SIP method, PUBLISH, that can be used to update the state of the Resource and this RFC is what we will be discussion in this chapter.

[TODO – insert figure. The figure should be so that Alice uses SIP but when Bob updates his state it is only done by some "other" means…]

Figure 4-1 illustrates how Alice subscribes to Bob's presence state and how he updates his state.

4.1.1 General Framework (again)

It is important, once again, to realize that the SIP event notification mechanism that we discussed in previous chapter is a generic framework and as such, RFC 3903 is also a

general mechanism for updating the state of a Resource. After all, that Resource is part of the generic framework.

4.1.2 Event State

The event state is the state information for a particular Resource for a particular Event Package. This state could be modified by and Event Publication Agent through issuing publish requests that are processed by an Event State Compositor who then will in turn update the event state of the particular Resource in question.

4.1.3 Event State Compositor (ESC)

The Event State Compositor (ESC) is responsible for processing incoming publish request and respond back to the publisher with an appropriate response. If successful, the published state will affect the Resource's state and if this state change should be conveyed to the Subscribers of that Resource, the Notifier will send out those necessary notifies.

If there are multiple publications made towards the same resource the ESC is also responsible for merging those published states into one unified view of the Resource. Multiple publication sources are common in e.g. the presence event-package and an example is when you leave your computer at home on during the day and then start another client on your work computers, both signed in to the same account.

Since the ESC cannot know the rules about how to merge multiple publication sources into one unified view for a particular Event Packages, each Event Package must also define how multiple PUBLISH request are combined into one unified state, the so-called composition policy.

4.1.4 Event Publication Agent (EPA)

A SIP client (UAC) capable of issuing PUBLISH requests is called an Event Publication Agent. This is a general term and sometimes when discussing a client that issues publish requests for a particular event package, such as presence, then a more specific term may exists.

4.1.5 Event Hard and Soft State

Event State that is installed by an EPA is usually referred to as soft state since as soon as the EPA goes offline, it will delete its state. Hence, soft state is usually installed by end clients and is volatile. However, it could also be possible to have persisted state that is always around. It could be the default state of a particular Resource for a particular Event Package and perhaps the best example is, once again, the presence case. Imagine that you go on vacation and will not have access to the internet for some time. You then could store your "offline" message stating that you are on vacation and people, Subscribers, subscribing for your presence state will see this "offline" message, i.e. your Event Hard State.

4.1.6 The PUBLISH Request

RFC 3903 introduces a new SIP method, namely the PUBLISH request and it is through this method an EPA can alter the Event State of a Resource. In contrast to the Subscribe request, the publish request does not establish a SIP dialog and therefore there will be no long lived peer-to-peer relationship between the two SIP endpoints (i.e., the endpoint issuing the publish request and the server who terminates it). However, there still exists an application level relationship between the two endpoints so even though there is no such thing as a subsequent publish request from a pure SIP perspective, it certainly exists from an application point-of-view.

4.2 Installing Event State

In Figure 4-1 we saw at a high-level how Bob changed the state of his presence state but did not discuss how this was done using SIP. We now know that this is done by constructing and sending a PUBLISH request that the ESC will process and a typical initial PUBLISH request is outlined in Listing 4-1.

An important part of this request is of course the actual state of the Resource that we wish to install. The new state is carried in the body of the request and whenever you include something in the body of a request in SIP, you also need to specify its format, which is done through the ContentType-header.

Of course, we must also tell the ESC which event-package and which Resource whose state we are about to change. Just as in the case of the subscribe request, this information is conveyed through the Event-header and the request-uri respectively.

```
PUBLISH sip:bob@example.com SIP/2.0
To: sip:bob@example.com
From: sip:bob@example.com;tag=857476
CallId: 98732847
CSeq: 1 PUBLISH
Event: presence
ContentType: application/pidf+xml
ContentLength: xxx

<XML body goes here, i.e. the state we are publishing>
```

Listing 4-1 shows an initial publish request.

If the request is accepted by the ESC the publisher will receive a 2xx response back indicating the successful installment of the published information (the published document) and if there were any Subscribers to that Resource they would have gotten new notify requests indicating the change of state (depending on local policy of course).

[TODO – insert the typical call-flow again but now with the PUBLISH request]

Figure 4-2 illustrates the typical call-flow

Note

All event packages known to me at the time of writing, the published state is conveyed as an XML document carried in the body of the request. Therefore, the act of publishing will from now on be referred to as installing a document in the ESC and it is that document that contains a part of the state of that Resource

4.3 Keeping the Event State alive

As with any state installed on a remote host it will expire if not refreshed within time and this is also true for the published document. Just as in the case of the subscription, the expiration time for publication is indicated in the 2xx response and the publisher needs to refresh the published document within this time. Looking from an application perspective, refreshing the publication is no different then refreshing the subscription but there is one big difference between the two cases and that is how do we create this "sub-sequent" request since the PUBLISH does not create a dialog? Well, the 2xx response contains yet another vital piece of information apart from the expiration time namely a unique identifier for the published document.

4.3.1 The SIP–ETag and SIP-If-Match headers

When manipulating the published state, the published document, in any way you will always be receiving a "receipt". If you later want to manipulate that particular document again you will need to present that "receipt" so that the ESC knows which document you are referring to. This receipt is conveyed through the SIP-ETag header as shown in the 200 OK response in Listing 4-2.

```
TODO – insert 200 response here
```

Listing 4-2 shows the 200 OK response to the initial publication in Listing 4-1

In the "sub-sequent" publish request, this receipt is inserted into the SIP-IF-Match header and if correct, the ESC will be able to find the document and will send back a 2xx response. Note, the 2xx response will actually contain a new unique identifier so you cannot use the same ETag twice. If the value in the SIP-IF-Match header is not recognized by the ESC it will send back a 412 Condition Failed response so if you happen to loose your receipt you are out of luck, there is no way to get it back and the state you installed will eventually time out and be removed.

Finally, how do we refresh the state we published initially? Easy, simply create a new PUBLISH request and insert the value from the SIP-ETag into the SIP-IF-Match header. Since we are simply resetting the timer associated with this state we do not need to include a body this time around and the refresh publish request is outlined in Listing 4-3.

```
PUBLISH sip:bob@example.com SIP/2.0
To: sip:bob@example.com
From: sip:bob@example.com;tag=857476
CallId: 98732847
CSeq: 2 PUBLISH
Event: presence
Expire: 3600
SIP-IF-Match: 98wuroisjlkasjdf

TODO – check the SIP-IF-Match spelling
```

Listing 4-3 shows the refresh publication.

Note that we this time also have included an expiration value, which we did not do in the initial publish request in Listing 4-1. Just as in the case of the subscription, if a time is not suggested in the request the ESC will choose one for you and that expiration time will always be conveyed in the 2xx response. Remember, whatever you put into the Expire-header in the request it is merely a suggestion and the ESC can really choose whatever it wants (with some restrictions) so you will always have to examine the 2xx response. Listing 4-4 shows the 200 response to the refresh publish request above and notice how the value in the SIP-ETag is a new unique identifier and not the same as the one we just inserted into the SIP-IF-Match header.

```
Yet another 2xx reponse and of course a new SIP-ETag, which is
what we will use in the next sub-sequent request.
```

Listing 4-4. The 200 OK response to the refresh publication.

[TODO – perhaps I should include a call flow here pointing out that the refresh does not ever cause notifies to be sent out?]

4.4 Updating the Event State

A common operation is to update the previously installed state and this is done by issuing an update publish request. This request is no different than the refresh request except that we now also need to include a body with the new state. Hence, the only difference between an update and a refresh publish request is that one has a body, the other one doesn't.

```
PUBLISH sip:bob@example.com SIP/2.0
To: sip:bob@example.com
From: sip:bob@example.com;tag=857476
CallId: 98732847
CSeq: 3 PUBLISH
Event: presence
Expire: 3600
SIP-IF-Match: <whatever the previous one will be>
ContentType: application/pidf+xml
ContentLength: xxx

<XML body goes here, i.e. the state we are publishing>
```

Listing 4-5 shows the update publish request.

Since updating the event state means that the overall state of that Resource may change, which may lead to notifications being sent out to its Subscribers. The reason why the updating of the state only "may" lead to NOTIFY:s being sent out is because the overall state of the Resource may not change due to the composition policy and even if it did change the state of the Resource, the subscribers may not receive notifications anyway because of authorization policies etc. We will discuss composition and authorization policies in greater detail in section XXX.

4.5 Deleting the Event State

A well-behaved client will have to delete its state at some point. In the case of presence, this usually happens when the client is shut down and if the client did not delete its state, the subscribers to this Resource would get the wrong state. Deleting the state involves once again creating a sub-sequent request and the request looks exactly the same as the refresh request except that the expiration time is set to zero, as the request below shows.

```
PUBLISH sip:bob@example.com SIP/2.0
To: sip:bob@example.com
From: sip:bob@example.com;tag=857476
CallId: 98732847
CSeq: 4 PUBLISH
Event: presence
Expire: 0
SIP-IF-Match: <whatever the previous one will be>
```

Listing 4-6 the publish request that deletes our previously installed state.

Of course, as with any request we need a response and if all went well we should be getting a 2xx response back and this response will actually also contain a new SIP-ETag, an identifier we actually never will be using.

Deleting the published state for a particular Resource also means that we do change parts of the state of the Resource. However, as previously mentioned, the overall state of the Resource may or may not change and even if it did change, due to local policies, the Subscribers may or may not receive notifications.

[TODO – why the new etag really? Just consistency or some other thought behind it? Check it]

4.6 Notifier + ESC = Event Notification Service

Once again, remember that this is a general system and it only provides a framework for implementing event packages. The mere framework by itself actually doesn't do anything (hence the name framework) but relies on a particular event package to define the context. One of the two important parts that an event package must specify is how

authorization is to take place as well as how multiple publications made to the same resource is handled (the composition policy).

4.7 Details

4.7.1 The “sub-sequent” PUBLISH request

We have seen that as soon as we create a “sub-sequent” publish request we will have to include the unique identifier we obtained from the last 2xx response for that particular document. This “receipt” is passed on to the ESC through the SIP-IF-Match header so that it will be able to locate the particular document we are targeting. However, there are more guidelines that should be followed when creating those “sub-sequent” publish requests and they are actually the same as for the SIP REGISTER request [RFC3261] and those are:

- The Call-ID should be the same
- The CSeq should be incremented by one for each new publish request.
- TODO – what about the local-tag?

An ESC should be able to handle the case where these guidelines are not followed but the EPA SHOULD follow these rules. Also, it makes it much easier to trace all the publish requests in a log file when you sit and debug a call-flow.

4.7.2 503 to quench traffic, doesn’t really work! See bugdb

discuss, this… there must be someone else out there who has had a similar experience.

4.8 Summary

Now that we have defined a complete framework where it is both possible to subscribe to state changes in a Resource as well as a way for a Resource to be updated by external means,

Through the help of the SIP PUBLISH request we now know how we can accomplish an external state change of a particular Resource.

Now that we have gotten yet another important piece of the puzzle down we do have the very fundamental knowledge of how an event notification system that also allows external updates over SIP to take place we are ready to actually start looking into the Presence Event Package.

CHAPTER 5

Presence Event Package

The previous two chapters did introduce a general framework using SIP as the underlying protocol in order to provide applications with a tool-box offering a way of registering an interest in a Resource and get notified when the state of that Resource changes. However, this framework does need additional specifications to define exactly what to do with the published information etc and this is defined by an Event-Package.

The most common event-package at the time of writing is the Presence Event Package. Anyone who has used any type of instant messaging client has come across presence in some form or shape. All the popular instant messaging clients have the ability to specify if you are available, busy or away and this is presence information, i.e. your willingness to communicate.

5.1 Introduction

Presence as a model was introduced by RFC 2778 and this specification discusses the basic idea of presence, sets the terminology as well as specifies the responsibilities of the different entities in a "Presence Service". This RFC does not concern itself with how this is implemented or which protocol is used to make this happen. Without knowing it, we have been discussing this model throughout the book but so far we have only been using general terminology. Since it is important to also know the "presence" terminology we will discuss them here.

Note

A "Messiging Service" is also defined in RFC 2778 but since this book is only concerned with presence, this service will not be discussed.

5.1.1 Presence Service

The Presence Service is responsible for accepting, storing and distributing presence information. Presentities are what feed the Presence Service with information, which then is sent out to Watchers. The role of what the Presence Service is what we previously have seen in the Notifier and Event State Compositor combined.

5.1.2 Presentity

A Presentity (presence entity) provides Presence Information to the Presence Service and as such it would be easy to map the Presentity to same role as the Publisher. However, it turns out that this is not as easily done as one would think. RFC 2778 seems to describe the Presentity as the one single source of truth of the presence state for a particular entity, such as Alice or Bob. The Publisher, on the other hand, is in fact only contributing with a part of the overall state of the Resource. In the general framework, it is the Resource that represents Alice and contains her combined state, a state that immerged from all the information published by the various Publishers representing her in some way.

As such, in general terms the Presentity is really the Resource and what RFC 2778 describes is really a Presence Publisher. As we will see when we discuss the Presence Event Package [RFC3856] in detail, this definition is inline with that RFC.

[TODO – think about this. Read 2778 and make sure that this conclusion is in fact correct. I am not sure it really is actually. Check out chapter 3 – Model in that RFC since it contains some more info. Also read 3856 and compare, maybe it sheds some light on this…]

------- 8< ----- cut here unless we find some need for it ----

Imagine that Alice has both a mobile phone and a work computer, both to which she is signed in. As such, she is

It is true that in RFC 2778 it is described as the Presentity feeds the Presence Service with information,

However, there is a slight deviation between the term Presentity and the general Publisher and even the Presentity defined by RFC

However, there is one slight deviation between the term Presentity as discussed in RFC 2778 and the general Publisher and actually the Presentity discussed in RFC 3856 as well. RFC 2778 seems to indicate that a Presentity is one single entity which provides the whole truth about the presence state to the Presence Service. However, a Publisher is in fact only contributing a small part to the overall state of the Resource

----- 8< ----------

5.1.3 Watcher

The Watchers are receiving state updates from the Presence Service whenever the Presence Service finds it necessary to send out updates. There are two main different types of Watchers, namely the Subscriber and the Fetcher. The Subscriber is an entity

that will register an interest with the Presence Service in receiving state updates for a particular Presentity. As such, the Subscriber will be receiving updates whenever the state changes of that Presentity changes. However, the Fetcher asks for the state explicitly. A special case of the Fetcher is the Poller, who fetches the presence information on a regular basis.

All of these types of Watchers should sound very familiar. In fact, the term “watcher” is just the special “presence” name for the term Subscriber that we introduced in CHAPTER 3 (one type of Watcher is even called Subscriber!). In section 3.5 we even discussed how to accomplish polling and as such, when a Subscriber does a poll they in affect become a Poller (or Fetcher depending if it is done on a regular basis).

5.2 The Presence Event Package

5.3 The Presence Format – PIDF

5.4 Summary

We finally got the presence event package in place and saw how we could deploy that onto the generic Event Notification Service and as such actually achieve a first version of a system capable of handling “presence”. However, presence is of a sensitive nature and the Presentity must get a chance to decide whether Alice is allowed to see his presence or not. Privacy and policy is quite important to many event packages and this is very true also for the domain of presence. The next chapter deals with these issues.

[TODO – probably will remove the below since the next chapter kind of changed directions… or rather it was merged with the actual policies as well]

The first problem is to figure out how the Presentity can found out that someone just subscribed to him I.e., how does Bob find out that Alice requested to see his presence? Luckily, others have already figured out this problem as well and this is what we will discuss in the next chapter.

CHAPTER 6

Privacy

Privacy is often a big concern in many systems and presence is definitely one of those services that need to have a well-defined and solid way of enforcing authorization. In the previous chapters we have built up a system where it is possible to subscribe to the presence state of Bob and as soon as he changes his state, you will be notified. However, we have also realized that due to the sensitivity of the information requested by Subscribers in general, we need a way to control who has the right to subscribe to what. I.e., we need to provide Bob with a way of authorizing Alice to view his presence state and also a way for him to actually find out that Alice did request to see his information.

[TODO – another version here, not sure I want to keep it…] Not only must there be a way of expressing your privacy settings but unless there also exists some way of actually finding out that others have requested the state information about a Resource, we will not know when there is a need to update our privacy policy.

[TODO – might remove this now since it is kind of included in the first paragraph in next section] The first mechanism that we need to provide is a way for Bob to actually be able to find out that Alice just subscribed to his presence state.

6.1 Watcher Information

Watcher information is distributed to a Presentity by the Presence Service and its purpose is to convey to the Presentity what Watchers knows about its existence [RFC2778]. In other words, through the help of watcher information, Bob can find out if Alice has requested to see his presence state and act accordingly. The ability for Bob to actually find out if someone requested his presence state is a first important step and is needed so that he can take appropriate actions such as allowing Alice to see his presence state. Or why not block her from seeing his state.

RFC 3857 specifies a watcher information (winfo) event template-package that is to be deployed onto the general event notification framework. Since it is a template-package it needs to operate within the context of another event package and its purpose is to monitor the underlying event package for changes in subscription state. Hence, when the state of a subscription associated with a particular Resource in the underlying event package changes, the winfo event package will detect this and act appropriately.

6.1.1.1 The Inner…

Since it can be very confusing to know which Resource, which Subscriber and which Subscription we are referring to when discussing the winfo event package, RFC 3857 makes the suggestion to use “winfo” as a prefix to all those terms when talking about e.g. a Subscriber of the winfo event package. As such, without this qualifier we will always be referring to the inner event package, i.e. the event package that the winfo is monitoring.

6.1.1.2 Event Package Name

RFC 3265 mandates that any event package being deployed within this framework must define an event package name, a name that will be pushed into the Event-header of the Subscribe request. Since this is a template-package it only defines a suffix that is to be appended to the event package it monitors. This suffix is “.winfo” and when it e.g. sits and monitors the presence event package the Event-header would therefore be “presence.winfo”.

6.1.1.3 Expiration time

Like any other event package, the winfo event template-package should suggest a default expiration time to use for the winfo Subscriptions. However, since its purpose is to sit and monitor another event package it is suggested that the winfo Subscription’s expiration time should match whatever default time the inner event package mandates. Hence, there is no default suggested expiration time for this event package.

6.1.2 Subscribing

Just because the winfo event package happens to be a template-package doesn't make it any different from a SIP point-of-view. As such, the installment of the winfo Subscription is done in the same way as for any other type of Subscription and it is maintained and destroyed in the very same way. A notification is sent out whenever the state of the winfo Resource is changed and the state is carried in the body of the NOTIFY as what we would expect.

The main difference is really that the Resource in the winfo event package actually does not carry a state per say. Its state is really just a reflection of the state of the inner Subscriptions and whenever the state of those Subscriptions change, the winfo Resource reacts and may send out a NOTIFY as a result.

[TODO – not sure the last paragraph is really helpful at all]

6.1.3 State deltas

The winfo event package allows for partial states to be sent out to the winfo Subscriber. As such, it is important that it is possible to keep track of these partial states and therefore the state sent out will always contain a version number. If the state is found to be out-of-synch, the winfo Subscriber can ask for the full state to be sent out by issuing a subsequent SUBSCRIBE request [RFC3265].

6.1.4 Format of the NOTIFY

The format of the watcher information is specified in RFC 3858 and is an XML based format describing, of course, the state of the subscribers to a particular resource.

6.1.5 Typical Flow

The

[TODO – insert figure here]

Figure 6-1 illustrates the winfo event package in action.

6.1.6 Subscription State

We discussed the various states of a Subscription in section 3.6 and it is this very state that we are conveying through the winfo event package. However, the state diagram we discussed previously actually does not contain all the necessary states for us to have a working system. Imagine the following scenario:

In Figure 6-1 we say how Bob detected that Alice had subscribed for his presence state. However, what if Alice went offline before Bob would have come online? We know that when you un-subscribe to something, that subscription will be moved over to the terminated state and removed from the system. As such, when Bob finally gets his winfo subscription up and running, Alice will no longer be there. Hence, Bob would not find out that Alice, at some point before he got online, actually requested to see his presence state. This could really go on forever and one of the reasons we decided that we would

need some way of detecting if someone had asked to see e.g. your presence was so that we could take authorization actions to allow/disallow that particular person. As such, we need a way to prevent this.

6.2 Common Policy

Now that we know how Bob actually can find out if someone requested to see his presence we need to decide on how his decision can be expressed in a way that the Presence Service can act upon it. As with everything else we have been doing so far, there is also an interest to create a common framework since the ability to express privacy is useful to many other event package as well.

Bibliography

[Blinn76] Blinn, J., F., Newell, M., E. (1976). *Texture and Reflection In Computer Generated Images*. CACM, 19(10), pages 542-547

[Bourke01] Bourke, Paul (2001). Texture map correction for spherical mapping.

http://astronomy.swin.edu.au/~pbourke/texture/polargrid

[Catmull74] Catmull, Ed (1974). *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD Thesis, University of Utah.

[Davis99] Davis, Tom et. al. (1999) OpenGL Programming Guide, 3rd Edition – The Official Guide to Learning OpenGL, Version 1.2

Addison-Wesley

ISBN: 0-201-60458-2

[EVL91] Electronic Visualization Laboratory at the University of Illinois at Chicago

http://www.evl.uic.edu/

[EVL94] Electronic Visualization Laboratory at the University of Illinois at Chicago

http://www.evl.uic.edu/

[Green86] Green, Ned (1986). *Applications of World Projections*. In proceedings of Graphics Interface ’86, pages 108-114, May 1986

[Heckbert86] Heckbert, Paul, S. (1986). *Survey of Texture Mapping*. IEEE Computer Graphics and Applications, Nov. 1986, pages 56-67.

[Heidrich98] Heidrich, Wolfgang, Seidel, Hans-Peter (1998). *View-independent Environment Maps*. In proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware, pages 39-45, 1998.

[NVidia99] NVidia Corporation (1999). Perfect Reflections and Specular Lighting Effects With Cube Environment Mapping. Technical Brief.

[Sweet00] Sweet, Michael, Wright, Richard, S. (2000) *OpenGL Super Bible, 2nd Edition*

Waite Group Press

[IMAX00] IMAX Corporation (2000).

http://www.imax.com

[Zimmons99] Zimmons, Paul (1999). Spherical, Cubic and Parabolic Environment Mappings.

http://www.cs.unc.edu/~zimmons/cs238/maps/environment.html