H. Schulzrinne
R. Shacham
Columbia University
W. Kellerer
S. Thakolsri
DoCoMo Eurolabs

June 25, 2006

Composing Presence Information
draft-schulzrinne-simple-composition-02

Status of this Memo

Copyright Notice

Abstract

Composition creates a presence document from multiple components published by one or more sources. This document identifies sources of information that a compositor might draw on presence composition and describes steps for composition. The composing function can be complex, so we intentionally restrict the discussion to cases that are likely to be common across many users of presence systems. We present an XML format for specifying a composition policy, based on our discussion.

Table of Contents

1. Introduction

Composition combines multiple presence or event sources into one view, which is then delivered, after various filtering operations, to watchers [6] [7]. Composition is required whenever there are several sources contributing information about a single presentity or event.

[Note: The content in this draft overlaps with the Processing Model draft and needs to be reconciled. Here, the emphasis is on developing the foundations for a composition policy language, and deal with merging <person> tuples.]

For notational simplicity and since most of the discussion has focused on presence rather than general events, we will restrict our attention to presence information using the Presence Information Data Format (PIDF) [3] and extensions such as the Rich Presence Information Data format (RPID) [4], keeping in mind that other types of events or status may well be able to use many of the same mechanisms. We assume that a presentity is a single human being. There are other presentities, such as the collection of customer service agents in a call center, where consistency is much harder to define.

We assume that the composition operation does not depend on the watcher identity, as there seems little functional gain by introducing per-watcher composing operations. The composed document contains the maximum set of information, i.e., no watcher can obtain more information than is contained in the composed raw presence document. (In some cases, a presentity wants to "polite block" a person by providing presence information that offers no information to the watcher, but avoids indicating that the watcher's subscription request has either not yet been processed or that it has been turned down. For those cases, a simple template that reflects a minimal PIDF document is sufficient, as it does not need to reflect presence inputs and does not change over time.)

Composition at the presence agent is just one component of providing useful and correct information to the watcher. We assume that composition is algorithmic, although manual composition by the presentity is theoretically possible. Given the automated nature of composition, there may well be situations where the best course of action is to expose the underlying data to the watcher, even though it may be contradictory. Indeed, in many cases, a mechanical composer may not even be able to detect whether information is contradictory or not.

The goals of composition are to remove information that is either stale, contradictory or redundant, to generate inferred presence

state and to represent presence information in a more useful way. Stale information has been superseded by other, newer information. Contradictory information makes two statements about the presentity that cannot both be true. Redundant presence information provides information that is no longer of interest. For example, a presentity may decide to drop information about services whose status is closed if there are open services and may drop a service record referring to another person via a <relationship> element if the presentity itself is available. Inferred presence state uses presence elements or external information to derive new information. Location information seems particularly suitable for such inferences. For example, a location away from home might generate the activity indication 'away' or specific geospatial locations might be mapped to particular location types or activities. Presence information may be presented in a useful manner by merging non-contradictory information.

Composition is not designed to reduce the size of notification messages or to protect information for privacy. Various compression schemes and partial notification [10] are better suited to reduce message sizes. Privacy filtering [8] has the role of tailoring information to individual recipients, based on the presentity's privacy policy.

In our model, the composer is reactive. In other words, it only creates a new presence document if one of the publishers updates parts of the presence document. An active composer could, for example, generate a new presence document after a certain time interval has elapsed or when timed presence [5] information is transitioning from the future to the presence.

The goal of this document is to outline options and then to derive a composition policy language that allows the user to control the steps that produce his presence document according to the aforementioned goals. Alternatively, a presence composition language can focus on the XML document and its components. Such a general presence composition language would have to be a full programming language, as it would need to support standard programming constructs such as conditionals, operations on XML elements in a document object model, history and external sources. This document focuses on content-aware policies rather than simple tools for mechanical transformations of XML presence documents.

2. Types of Information sources

Presence information can be contributed by many different sources, either directly, by publishers using PUBLISH requests or by a presence agent acting as a watcher receiving NOTIFY requests. We

describe each mode of delivery operation in the following. In direct mode, the composer has direct access, without presence protocol mediation, to this information, e.g., via REGISTER requests or layer-2 operations or access to user keyboard activity. Secondly, sources can use SIP PUBLISH requests to update presence information. Finally, presence agents can in turn subscribe to presence information and receive NOTIFY requests. However, the mechanism of data delivery is likely to be less important than the original data source and how the information was derived. Thus, to the extent possible, information about the original source should be preserved as otherwise information might become more credible simply because it has been re-published. We focus here on the semantic source of the data, i.e., how it was derived, not how it was injected into the presence system.

For simplicity, we do not try to assess the veracity of the presence document. In order to evaluate the usefulness of a presence document, we only care whether the presentity would want the information to appear that way, not whether this corresponds to observable facts. Thus, a presence document is correct in that sense if it indicates that the presentity is in a meeting even though the presentity has actually gone fishing if the presentity would like the rest of the world to believe that he is at work. It may, however, well be the case that composition policies find it easier to maintain the truth than keep lies consistent across sources of presence information.

We can distinguish the following sources of presence data:

Reported current: Reported current information has been provided by the presentity within processing time delays of the current time. A presentity can update status information manually, by setting any of the element in a presence document. This update may be made by sending a PUBLISH request, by using XCAP as specified in [11] , or by a more direct update, such as editing it in a web GUI. We assume that this information is correct when entered, but the trustworthiness of the information is likely to decay as time goes on, given that most human users will find it difficult to continuously keep presence information up-to-date.

Reported scheduled: For reported scheduled information, a presentity indicates its plans for the future rather than the present, e.g., in a calendar. The reliability of this information depends largely on the diligence of the user in updating calendars and similar sources.

Measured device information: Measured device information uses observed user behavior on communication devices, such as the act of placing or receiving calls or typing. The main source of error is that a device may not be able to tell whether the presentity itself is using the device or some other person.

Measured by sensors: Presence information measured by sensors reflects the status of the presentity, e.g., its location, type of location, activity or other environmental factors. Examples of sensors include Global Positioning System (GPS) information for location or a BlueTooth beacon that announces the type of location, such as "theater", a person finds itself in. Sensors have the advantage that they do not rely on humans to keep the information up-to-date, but sensors are naturally subject to measurement errors. In particular, in quantum mechanical fashion, it is sometimes difficult to ascertain both the measured variable and the identity of the presentity. For example, a passive infrared sensor (PIR) can detect that somebody is in the office of the presentity, but cannot detect whether this is the presentity himself, cleaning staff or a dog. A GPS sensor cannot detect whether the cell phone is being used by the presentity or has been borrowed by the presentity's spouse.

Derived: Presence information might be derived indirectly from other sources of data. For example, the basic open/closed status might be algorithmically derived from a variety of other, watcher-visible or not, elements.

3. Composition Steps

In our model, presence takes a presence document, made up of a set of <tuple>, <person> and <device> tuples, each tuple consisting of one or more elements, and creates another valid presence document based on this information. Based on the aforementioned goals of removing stale, contradictory or reduntant information, while providing additional useful data and representing the information in a useful manner, our model includes a sequence of operations on the input tuples. These operations are: discarding, derivation, conflict resolution and merging. Discarding tuples removes stale and redundant information. Derivation provides useful new data. Conflict resolution removes contradictory information. Merging presents the presence in a useful manner.

Composition involves adding or removing information from a set of sources, and this may be done at a tuple or element granularity. Some of the steps operate at one granularity or another. While any of the operations may be done on any tuple type, some operations may be more likely performed on certain types. This information is summarized in Table 1. Each of the steps is listed, along with the

granularity on which it typically operates, and whether it is likely or unlikely to be used for each of the tuple types. The specific elements in the table will be discussed in later sections.

Operation	granularity	<person>	<tuple>	<device>
Discarding	tuple	likely	likely	likely
Derivation	element	likely	likely	likely
Conflict Resolution	tuple or element	likely	unlikely	unlikely
Merging	element	likely	likely	unlikely

Table 1

4. Discarding

Whole tuples may be discarded based on zero or more of the criteria below:

Closed contacts: All <tuple>s with a basic status of 'closed'.
Old tuples: Tuples (<person>, <tuple>, or <device>) whose age is older than a given threshold. Since presence information should be automatically removed after its expiration time, this discarding applies only to tuples before their expiration.
Unreferenced tuples: <device> tuples that are not referenced by any service <tuple>. (It should be noted that user activity information about these devices may still be useful even if the device itself is not part of any published service.)

5. Deriving Presence Information

Certain presence sources may not be capable of publishing all relevant information, and users are unlikely to always update all information that requires their input. Such information may be derived in order to include it in the presence document.

Derivation of new information makes it easier to identify a conflict with another presence source. For example, knowing the locations of two presence sources allows the compositor to determine that the user is only colocated with one of them, and the information from the other one is inaccurate. It can also provide information to the watcher indicating communication capability that may not otherwise be known. For example, a user's mobile device may easily be able to identify and publish that it is in a car. However, more relevant

information for the watcher is that the user is driving, which may be derived if this is usually true when the user is in a car (possibly during certain times, such as weekday mornings and evenings). The user may also wish to indicate that when he is "on-the-phone" (which may be published automatically by the UA once he has successfully set up a dialog), this means that he is "busy" and shouldn't be called except in an emergency. The user may know that a specific place does not allow for private communications, and he may automatically supplement his location information with privacy information. More complex rules could be derived that involve outside information such as time of day. For example, when user-input is "idle" between certain hours of the night, the user's activity should be set to "sleeping".

Such derivations each have a predicate for defining the conditions of the derivation, and an addition of XML content. The predicate is one or more elements that must all be present in a tuple in order for the content to be added there.

A special case of this is the supplementing of static information that doesn't depend on dynamically changing predicates. For example, a device may not support RPID extensions, but they may be added to its presence tuple and that of its associated service using derivation. Such a derivation would be declared using a specific value for the contact address or device-id as the predicate (for a service or device, respectively).

There is another way that this static information can be supplemented. The XCAP mechanism described in [11] is used for updating a user's presence. XCAP does not manipulate the user's complete presence document, but, rather, a single presence document which is one of the sources input to the compositor, along with information sent by other presence sources, through PUBLISH or event notifications. XCAP may be used to create <tuple> and <device> tuples containing static information about the service or device. During the composition process, multiple reports for a single service (those containing identical <contact>s) and for a single device (those <device> tuples containing identical <device-ID>s) are merged together. If no identical <tuple> or <device> tuple has been received from any other source, the static tuple will appear in the resulting raw presence document. If there is another identical tuple, the static and dynamic elements will be merged into a single tuple. The <basic> status of any service appearing in the XCAP document should be "closed" so that this becomes the default status and, when the service is published by another source with a status of "open", the resulting status will be "open", which is the union of the two. It should be noted that the technique described here is predicated on the merging of service <tuples>s, which we are

currently leaving out of our model as discussed in Section 9.4, and plan to specify in the future.

6. Resolving Conflicts

6.1. Sources of Information Conflict

Information conflict occurs when multiple sources give different views of the presentity, some of which may be outdated or incorrect. Information can be incorrect for any number of reasons, but some examples include:

Location divergence: The publisher collecting the information may not be colocated with the presentity at this particular time. For example, Alice's home PC may report that the user is idle (not typing), but Alice is using the office PC.

Update diligence: Some sources, particularly those updated manually, are prone to only approximate reality. For example, few users record all appointments or meetings in their calendar or, conversely, remove all canceled meetings. This is particularly true for regularly scheduled activities such as meals or commute times.

Sensor failure: Sources that report their information differentially are subject to silence ambiguity. If such a source does not report new data, the receiver cannot tell whether the sensor is malfunctioning or whether the information last received is still current. This can be partially mitigated by requiring sources to report when they are no longer confident of the data. However, this does not deal with sudden source failures. Thus, some form of keep-alive mechanism may well be needed that overrides differential notification mechanisms. Even with keep-alive, there is likely to be a substantial period of time between source failure and failure detection, causing stale information.

6.2. Detecting information conflicts

We would like to be able to detect information conflicts, so that appropriate processing logic can remove inaccurate information. There are many elements in <person> tuples that could end up having conflicting values from different sources. However, this step is less relevant for service tuples. The elements found there are not likely to conflict, even if multiple tuples report information about the same service. For example, the basic status in a service tuple cannot be said to conflict with the status sent for a service on another device. In fact, for the static information derivation described in Section 5, the different values must not be treated as conflicting so that the tuples can be merged in the next step.

<deviceID>, <privacy>, and <user-input> describe a specific instance of the service and can all be true. Of course, if service tuples are merged as described in Section 7, the multiple values must be handled in some way, such as listing all of them or choosing one. Our discussion of conflict resolution is focused primarily on person information.

Information conflicts can be classified as to whether they are detectable in a single element or only across elements and how easy it is to detect them.

Single-element conflicts occur if two elements, say <activities> in RPID, in two sources cannot both be true or are highly unlikely to be true, without having to inspect any other element. A multi-element conflict occurs if only the combination of multiple elements indicates a conflict.

Multi-element conflicts often have location, and properties known for this location, as the common element. For example, certain geospatial locations are known not to contain certain types of places. Thus, both the location and the <place-type> information are, by themselves, each credible and possible, but are detectably wrong once considered together. These conflicts can be detected if location or time can be mapped to reliable information from external sources. As mentioned above, derived information can make conflict detection easier by supplementing information to create a single-element conflict.

We distinguish three types of information conflict: obvious, probable and undetectable, described in turn below.

For some pieces of presence information, information conflicts are obvious and readily detectable. For example, under the one-person-per-presentity assumption and common assumptions of physics, a single presentity can only be in one place at a time. Thus, if two sources report location information that differs by more than the margin of error, one must be wrong. In RPID, the <place-is>, <privacy>, <relationship>, <time-offset>, and <user-input> elements have exlusive values, although in some cases, below the element level. For example, the <privacy> field has information for both audio and video, and thus two sources may report different information for <privacy> and still both be correct as long as they refer to different media types.

For other types of information, an automaton can guess with some probability that two sources of information contradict each other, but this may well depend on the values themselves. For example, the <activities> combination of

away, appointment, in-transit, meeting, on-the-phone, steering

incrementally reported by different sources may well reflect the activity of the typical Wall Street commuter in the Lincoln Tunnel, speaking on his cell phone. One would hope, however, that combinations such as "steering, sleeping" are rarely true, although "sleeping, meeting" indicates that there are few activities that completely rule out others. The <place-type> element is another one that may take different values, sometimes, but not always, contradictory. For example, the values "outdoors" and "stadium" differ only in their specificity. For these types of elements, two options seem possible. A table may be constructed with each value in both a separate row and a separate column, so that their relationships may be charted. The relationship of value A to B may be contradictory, more or less specific, or have no relationship. Alternatively, different values may always be treated as contradictory. The latter approach seems better suited for an element like <place-type> where a single source is likely to have all relevant information and can be fully accurate by itself. However, this works less effectively for <activity>, for instance, where different sources inherently give different types of information. For example, a cell-phone says that the user is "on-the-phone", a sensor says the user is "steering", and a calendar says that the user is in a "meeting".

Undetectable information conflicts are those where a machine lacking human intelligence cannot reliable detect that the two pieces of information cannot both be true. For example, an automaton is unlikely to be able to decide which of several notes or free-text fields is valid, without basing this on other information in the tuple, person or device element.

6.3. Handling Information Conflicts

Once an information conflict is detected, a choice must be made about how to handle it. In some cases, no action should be taken. For an element such as <activities> or <mood>, for which different reported values makes sense and it is hard to distinguish which values really conflict, as mentioned above, the different values can be treated as non-conflicting. This means that both tuples are retained, and handling is deferred to the merging step, during which the multiple values will be unioned within a single tuple.

For other elements, however, conflict is more easily detectable and multiple values are not sensical. A conservative approach to handling such a conflict would be to simply list all values. This is different from the approach mentioned earlier, because the tuples are kept distinct and not merged in the next step. Multiple versions are

presented which are admittedly conflicting, and the watcher may make a judgment about which is more correct. To limit the amount of information that the watcher must digest, it may be more useful to choose one value over the other. For this decision, a number of common heuristics may be used, which are listed below:

Choose recent tuple: Choose the value from the tuple that was more recently published for the first time. Simply choosing the most recently updated value is likely to cause flip-flopping between dueling publishers.

Choose trustworthy tuple: Choose the element from the more trustworthy tuple. Trustworthiness may be based on the source identity, such as a user's cell phone. Alternatively, it is based on the types of reporting listed in Section 2. For example, they may be ranked in the order "reported current", "measured device information", "measured by sensors", "reported scheduled", and finally "derived".

Value of another element: Other elements may indicate that one version of the information should be trusted. For example, <user-input> may indicate that one device that provides presence is being used by the user, and another is not. As a special case of this policy, tuples belonging to a certain sphere may be given precedence. For example, after a certain hour, it is more likely that the tuple with the <home> sphere is up-to-date.

Specific heuristics may be combined with external information, such as time of day.

As new elements are added, they are likely to either fall into the category of elements where collecting all values makes most sense, such as activities and mood above, or where a choice among values needs to be made.

When one value is chosen over another, the resulting presence document may be affected on the tuple level or on the element level. On the tuple level, the more trusted tuple is chosen and the other is discarded. On the element level, both tuples are maintained, but only the more trusted element is kept, while the other is discarded.

Either of these approaches may have advantages in certain situations. However, we propose using only tuple-level conflict resolution to avoid inconsistencies in the final document.

7. Tuple Merging

Merging combines several tuples that logically represent the same information. For example, a presence document should only contain

one report of <person> information, so the multiple reports from different sources should be merged. It may also be useful to merge service <tuple>s that have the same contact URI. (We leave aside for now the difficulty of deciding whether two URIs that are not lexically identical are indeed functionally the same) This may occur when the same service is being provided by a variety of devices, or in the example of static information in Section 5. Sometimes, it is better not to merge tuples. For example, some elements can contain timing information indicating the range of time that the information is believed to be valid. It is probably not a good idea to combine elements that cover different, although maybe overlapping, time intervals.

In any of the above cases, the elements in the resulting tuple must be based on the original tuples. Although the original values should not conflict, following the previous step, some elements will have multiple non-conflicting values, when multiple services are merged or person tuples are merged which contain elements which are treated as non-conflicting, as described above. When this occurs, either element must be selected or they should be unioned. We discuss appropriate techniques for each element type below.

7.1. Service tuples

When composing <service> tuples, the following rules apply to their PIDF and RPID elements:

basic status: The union of all values should be returned, so that the service is 'open' as long as one source reports 'open'.

class: A single value needs to be chosen.

deviceID: If a service is offered by multiple devices, it makes sense to enumerate all the device identifiers.

privacy: Since the caller cannot select the device that satisfies specific privacy requirements, the appropriate choice is to provide the most conservative indication of the privacy to be expected, i.e., the least privacy indicated among all the tuples for the contact URI.

relationship: If two tuples with the same contact URI differ in their relationship, the relationship element needs to be dropped.

status icon: It is a local choice whether to present all status icons, as they may reflect specific capabilities, or choose one.

user input: In a combined <tuple>, it makes sense to reflect the most recent user input.

7.2. Person tuples

As noted in the section on conflict handling, there are elements for which different values may be treated as non-conflicting. These may include <activities>, <mood>, and <place-type>. For such elements,

all values are unioned in this step.

8. Default Policy

The default composition policy is designed to lose no information, at the expense of presenting possibly contradictory information to watchers.

This composition policy performs a union with replacement. Newly published elements replace earlier elements with the same 'id' attribute. We assume that each source chooses their own 'id' values.

Other than this, all elements are simply enumerated as is, sorted by type (person, tuple, device). Elements within the <person>, <tuple> and <device> elements are not modified at all, except possibly annotated with a source description (and timestamp?). This policy can also be seen as providing input to the following steps.

9. Composition Policy Format

We define an XML format for specifying a policy for composition. It is expected that this format will be used by users themselves, and that standard composition documents be created by network administrators. The document is a sequence of composition steps, each with its own options for customization. The steps are "discard", "derive", "resolve-conflicts", and "merge".

9.1. Discard step

This step allows for discarding of tuples. Three types of discarding may be specified: discard all service tuples with closed contacts, all tuples whose timestamps are older than a certain amount of time, and all device tuples not associated with a service.

9.2. Derive step

This step contains rules for deriving new information based on existing information. The XML Patch format [12] is used to express the derivation of new content, using the <add> element. The XML content following the <add> element is the new content to be added, while the derivation conditions are expressed in the 'sel' attribute of that element. This attribute takes as its value an XPath [13] expression which identifies the location where the content is to be added. Xpath predicates can be used to select only tuples with specific children, which forms the condition of the derivation expression.

For example, the following Patch operation:

```
<add sel='//person[place-type/car]'>
  <activities>
    <driving />
  </activities>
</add>
```

adds the 'driving' activity to any person tuple that shows the 'place-type' as 'car'.

In order to make derivation dependent on the time of day, the selecting Xpath expression may refer to the tuple's timestamp in the predicate. Functions built into Xpath 2.0 may be used to retrieve the desired part of the date/time expression. For example, if someone sleeps between the hours of midnight and 7 am unless he is working on a deadline, a derivation of his sleep based on his user-input may be expressed as follows:

```
<add sel='//person[user-input="idle"] \
[fn:hours-from-dateTime(timestamp) > 0 \
and fn:hours-from-dateTime(timestamp) < 7]'>
  <activities>
    <sleeping>
  </activities>
</add>
```

This states that if the user-input is 'idle' during normal sleeping hours, the user is sleeping. If the value is not 'idle' during those hours, he is likely working on a deadline.

9.3. Resolve Conflicts Step

In this step, conflicts are identified and resolved using one of a number of policies. Identifying conflicts is a matter of local policy as it is not seen as something that users should specify.

The <resolve-conflicts> element contains possibly several <conflict> elements, each defining how conflict is to be resolved. An "element" attribute may be included so that the included policy applies only to that element. When this attribute is omitted, or has a value of "all", it applies to all elements.

Options for resolution are "merge", "union", "most-recently-published", "source-precedence", or "other-attribute". Several policies may be listed, and conflict resolution is attempted with each in the order that they appear, until one succeeds.

<merge>, in effect, defines the given element as non-conflicting. Examples of elements appropriate for this are <activities> and <mood>. It is useful for this format be used to define these so that new presence elements may also be easily taken into account without requiring a configuration of the Presence Server. The use of <merge> for a given element precludes any other conflict resolution policy for that element.

Choosing "union" causes both conflicting tuples to be included, and precludes any other policy for conflict resolution for the specified elements. It also ensures that the two tuples will remain distinct, even after the merging step, so that multiple versions will be represented, and the human watcher will be able to decide which is more likely to be accurate. This is the default value for the resolution of a conflict for any given element when an alternative policy is not given.

The <most-recently-published> element directs the compositor to choose the tuple which was most recently published for the first time. This does not choose a tuple simply because it was refreshed more recently.

The <source-precedence> element lists a number of source types. This list may contain any of the following tokens at most once: "reported current", "reported scheduled", "measured device information", "measured by sensors", "derived". If each of the conflicting tuples is from one of the sources listed, the one with a higher value is chosen. If only one of the tuples is from a source with a listed value, that one is chosen. If neither of them are, the conflict is not resolved by this method.

The <other-element> element specifies that resolution be done based on another element besides the one in conflict. An attribute is included to specify the element. A list of elements gives the ordered preference of various values.

9.4. Merging

This final step merges multiple tuples to present a final view of the user's presence before continuing to later steps such as privacy filtering. We currently consider only merging of <person> tuples as this is the most likely to be useful.

When multiple tuples are merged, they may have different values for the same attribute. The conflict resolution step is used to declare for which elements, such as <activities> multiple values should be listed, rather than be treated as conflicting. Therefore, no real specification is required by the user in this step for <person>

tuples. It is expected that for the merging of service <tuple>s, input from the user will be desired regarding whether to merge them and, if so, how to handle multiple values of elements.

10. XML Example

```
<discard>
  <old-tuples age="00:30:00.000" />
  <tuples-with-closed-contacts />
</discard>
<derive>
  <add sel='//person[place-type/car]'>
    <activities>
      <driving>
    </activities>
  </add>
</derive>
<resolve-conflicts>
  <conflict element="activities">
    <merge />
  </conflict>
  <conflict element="mood">
    <merge />
  </conflict>
  <conflict element="place-type">
    <source-precedence>
      <source>reported current</source>
      <source>reported scheduled</source>
    </source-precedence>
    <other-attribute attribute='person/user-input'>
      <value>active</value>
      <value>idle</value>
    </other-attribute>
  </conflict-element>
</resolve-conflicts>
```

11. Security Considerations

Composition itself does not create new data types, although it might create new elements by derivation. Thus, the security considerations are the same as those for the constituent presence information elements.

12. IANA Considerations

This document does not request any IANA actions.

13. References

13.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Day, M., Rosenberg, J., and H. Sugano, "A Model for Presence and Instant Messaging", RFC 2778, February 2000.

[3] Sugano, H., Fujimoto, S., Klyne, G., Bateman, A., Carr, W., and J. Peterson, "Presence Information Data Format (PIDF)", RFC 3863, August 2004.

13.2. Informative References

[4] Schulzrinne, H., "RPID: Rich Presence Extensions to the Presence Information Data Format (PIDF)", draft-ietf-simple-rpid-10 (work in progress), December 2005.

[5] Schulzrinne, H., "Timed Presence Extensions to the Presence Information Data Format (PIDF) to Indicate Status Information for Past and Future Time Intervals", draft-ietf-simple-future-05 (work in progress), December 2005.

[6] Rosenberg, J., "A Data Model for Presence", draft-ietf-simple-presence-data-model-07 (work in progress), January 2006.

[7] Rosenberg, J., "A Processing Model for Presence", draft-rosenberg-simple-presence-processing-model-01 (work in progress), August 2005.

[8] Schulzrinne, H., "Common Policy: An XML Document Format for Expressing Privacy Preferences", draft-ietf-geopriv-common-policy-10 (work in progress), May 2006.

[9] Peterson, J., "A Presence-based GEOPRIV Location Object Format", draft-ietf-geopriv-pidf-lo-03 (work in progress), September 2004.

[10] Lonnfors, M., "Session Initiation Protocol (SIP) extension for Partial Notification of Presence Information", draft-ietf-simple-partial-notify-07 (work in progress),

June 2006.

[11] Isomaki, M., "An Extensible Markup Language (XML) Configuration Access Protocol (XCAP) Usage for Manipulating Presence Document Contents", draft-ietf-simple-xcap-pidf-manipulation-usage-02 (work in progress), October 2004.

[12] Urpalainen, J., "An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors", draft-ietf-simple-xml-patch-ops-02 (work in progress), March 2006.

[13] "XML Path Language (XPath) 2.0", W3C Candidate Recommendation 8 20060608, June 2006.

Appendix A. Acknowledgments

This document is based on discussions within the IETF SIMPLE working group. Paul Kyzivat provided helpful input.

Authors' Addresses

Henning Schulzrinne
Columbia University
Department of Computer Science
450 Computer Science Building
New York, NY 10027
US

Phone: +1 212 939 7004
Email: hgs+simple@cs.columbia.edu
URI: http://www.cs.columbia.edu

Ron Shacham
Columbia University
Department of Computer Science
450 Computer Science Building
New York, NY 10027
US

Email: shacham@cs.columbia.edu

Wolfgang Kellerer
DoCoMo Eurolabs
Landsberger Str. 312
Munich 80687
Germany

Email: kellerer@docomolab-euro.com

Srisakul Thakolsri
DoCoMo Eurolabs
Landsberger Str. 312
Munich 80687
Germany

Email: thakolsri@docomolab-euro.com

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.