Internet Engineering Task Force
Internet Draft

SIMPLE WG
J. Rosenberg
dynamicsoft
M. Isomaki
Nokia

draft-rosenberg-simple-data-req-00.txt
June 24, 2002
Expires: December 2002

Requirements for Manipulation of Data Elements in SIMPLE Systems

Abstract

In an instant messaging and presence application, it is frequently necessary for the user to configure a number of pieces of information. Users will need to manipulate their buddy list, adding and removing presentities, and manipulate their authorization lists, which specify the set of users that can subscribe to their presence. In this document, we provide a set of requirements for such data manipulations, and provide a framework for viewing them in a common way.

Table of Contents

1	Introduction	3
2	Buddy List Manipulation	3
2.1	Model	3
2.2	Requirements	4
3	Authorization Policy Manipulation	6
3.1	Model	6
3.2	Requirements	8
4	Manipulation of Feature Data	8
4.1	Model	8
4.2	Examples	9
4.3	Requirements	11
5	Possible Solutions	12
6	Authors Addresses	12
7	Normative References	13
8	Informative References	13

1 Introduction

Consumer-based instant messaging and presence applications typically provide a rich set of features. In addition to being able to subscribe to, and get notified of, changes in presence, users can also configure the operation of the application.

Most systems allow the user to add or remove users from their "buddy list". The buddy list is the set of presentities [1] that a user is subscribed to. This buddy list is frequently stored on the server, allowing the user to generate a single subscription to the entire list. The server then "fans out" that subscription too all the presentities on the list. Subscription to buddy lists is supported through the buddylist event package defined for SIMPLE [2]. However, no automated means is currently defined to create these lists, add users to them, remove users from them, or query for the set of users on the list.

Similarly, most systems support user-defined authorization policies. A user can specify which watchers are (or are not) allowed to subscribe to their presence, and furthermore, what aspects of their presence a watcher is able to see. While SIMPLE [3] systems can support such authorization policies, besides human-driven techniques, such as web or voice response, there is no automated way to specify these policies.

In this document, we propose a set of requirements for manipulation of buddy lists and authorization policies. We also provide a generalized framework for these problems, and present requirements for a generalized solution.

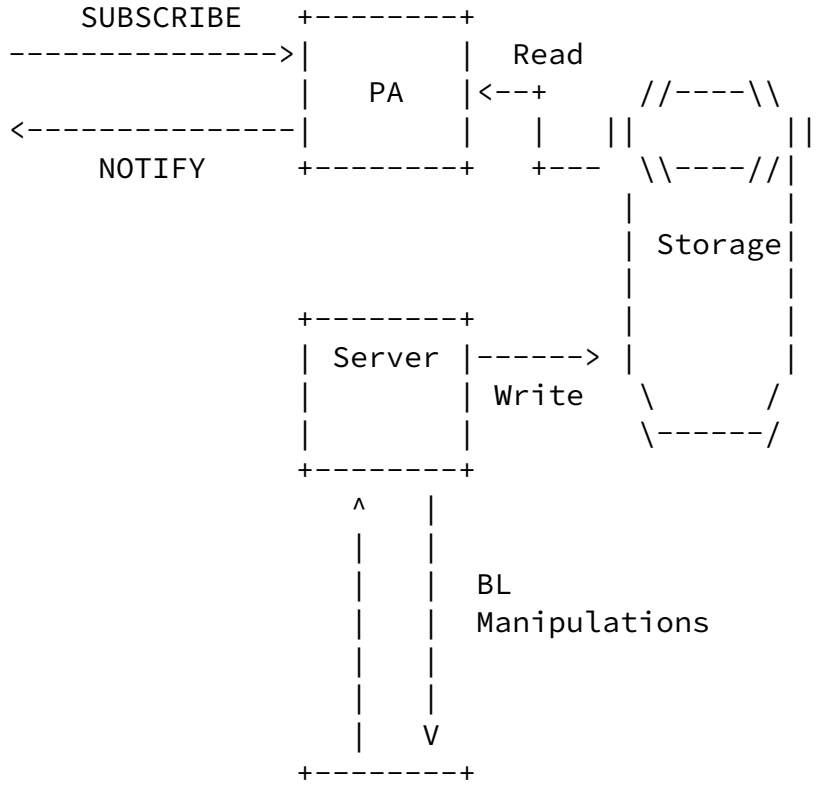
2 Buddy List Manipulation

2.1 Model

The model for the the usage and manipulation of a buddy list is shown in Figure 1.

A buddy list is defined as a set of presentities (each of which is represented by a URI). The buddy list is itself identified by a URI (for example, sip:myfriends@example.com). The SIP buddy list event package [2] allows a watcher to subscribe to the buddy list. Currently, buddy lists are manipulated through human interaction, such as on a web page or a voice response system. In order to support manipulation of the list by automata, protocol support is needed.

We assume that there is some kind of client-server protocol for such



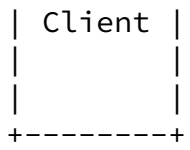


Figure 1: Model for Buddy List Manipulation

manipulation. The server stores the buddy list, and can directly manipulate it based on requests from the client. The presence agent (PA) can fetch the buddy list when it receives a subscribe request for it.

2.2 Requirements

The following are the set of requirements for such manipulations:

REQ 1: It MUST be possible for the client to create a buddylist and associate it with a URI.

REQ 2: It MUST be possible for the user to specify the URI for the buddylist when one is created. If the name cannot be allocated (because it already exists, for example), it MUST be possible to inform the client of the failure, and the reason for it.

REQ 3: It SHOULD be possible for the server to provide the client a URI for the list when one is created, in the case where the client does not provide it.

REQ 4: It MUST be possible to add an entry to the buddylist. It MUST be possible for the entry to be any URI that is meaningful in the context of a buddy list. Examples would include a SIP URI or pres URI [4].

REQ 5: It MUST be possible for a buddy list to contain entries which are themselves buddy lists.

REQ 6: It MUST be possible to remove an entry from the buddylist, by providing the URI for the specific entry to be removed. If the entry does not exist, it MUST be possible for the server to inform the client of this fact.

REQ 7: It SHOULD be possible to clear all entries from a buddy list.

REQ 8: It MUST be possible to delete a buddy list. In this context, deleted means that the name of the buddy list is no longer defined, so that subscriptions to the list would fail.

REQ 9: It MUST be possible to query for the set of URIs in a particular buddy list, by providing the URI for the buddy list.

REQ 10: It MUST be possible for the buddy list to be associated with a list of authorized users. Those authorized users are the only ones permitted to manipulate the buddy list.

REQ 11: It MUST be possible for a client to store a cached copy of the list. This implies that it MUST be possible for the server to notify the client of a change in the list. It MUST be possible for the client to manipulate the local cached copy even when there is no connectivity to the server. It MUST be possible to synchronize the cached copy

with the master copy on the server, when connectivity is re-established.

This particular requirement is crucial for wireless systems, where a copy of the list resides ont he handset. Without this requirement, a user would not be able to view the list, or add a user to it, when they go out of coverage.

REQ 12: It MUST be possible for there to be multiple clients with cached copies of the list.

REQ 13: Manipulations of the buddy list MUST exhibit the ACID

property; that is, they MUST be atomic, be consistent, durable, and operate independently.

REQ 14: It MAY be possible for the client to batch multiple operations (add a buddy, remove a buddy) into a single request that is processed atomically.

REQ 15: It MUST be possible for the server to authenticate the client.

REQ 16: It MUST be possible for the client to authenticate the server.

REQ 17: It MUST be possible for message integrity to be insured between the client and the server.

REQ 18: It MUST be possible for privacy to be insured between the client and server. As a motivating example, an eavesdropper on the protocol could ascertain the set of people on my buddy list, resulting in divulging private information.

3 Authorization Policy Manipulation

3.1 Model

When presence agent receives a subscription request, it makes a decision on whether the watcher is allowed to subscribe, and what they are allowed to subscribe to. The presentity can manipulate those policies, in order to support both off-line authorizations, and reactive authorizations (reactive authorizations are ones that are made in response to an attempt by the watcher to subscribe).

Similarly, when a proxy receives an IM, the proxy can execute policy which determines whether or not the IM should be forwarded to the

user.

Generally, there are two aspects to both of these policy systems. One is the logic that guides the policy, and the other is the data (such as lists of users) accessed by that logic. As an example, the logic might dictate that a watcher is checked against an explicit deny

list, and if present, their subscription is denied. If they are not on the deny list, they are checked against an explicit allow list, and if present, their subscription is accepted. If they are on neither list, they are marked as pending. This logic makes use of two lists, which represent the data.

In this model, the logic can be represented by a script, similar to the operation of a Call Processing Language (CPL) [5] script. The primitives of the scripting language would allow for access to the lists that represent the data. For example, a CPL-like script representing the policy example of the previous paragraph might look like:

```
<cpl>
  <subscription>
    <lookup source="sip:denylist@example.com">
      <success>
        <reject status="denied"/>
      </success>
      <notfound>
        <lookup source="sip:allowlist@example.com">
          <success>
            <accept/>
          </success>
          <notfound>
            <pending/>
          </notfound>
        </lookup>
      </notfound>
    </lookup>
  </subscription>
</cpl>
```

The deny and allow lists are, in this example, represented by SIP URIs. The script itself can also be represented by a URI. In order to activate a policy, a particular script is bound to the authorization function that executes at the PA (or SIP proxy server that would process an IM).

The overall architecture is, as a result, the same as is shown in Figure 1. The client manipulates the script and a set of lists at the server. The server stores this data, and it can be read by a presence agent in order to make an authorization decision.

3.2 Requirements

Based on this model, the following requirements can be specified:

REQ 1: It MUST be possible to bind a script defining the logic for processing to a particular authorization function.

REQ 2: It MUST be possible for the client to determine the set of supported scripting languages.

REQ 3: It MUST be possible for the server to reject the script because it is malformed, too complex, or not acceptable for some other reason.

REQ 4: It MUST be possible for the client to fetch the current script.

REQ 5: It MUST be possible for the client to indicate what script languages it supports when it fetches the script. In this way, a server could conceivably translate it to a format supported by the client.

Almost all of the requirements for buddy list manipulation as specified above also apply to manipulation of the script and of the lists. As such, we do not repeat them.

4 Manipulation of Feature Data

From a requirements analysis of the manipulation of buddy lists and of authorization policy, it is clear that there is a more general problem here. The problem is the manipulation of user feature data associated with applications. We therefore propose a model for this more general case, and specify requirements for it.

4.1 Model

In the proposed model, there is an application (also referred to as a feature) that is resident within the network. This application provides a value added service to the user. Sometimes, the user has control over the logic of the application itself. However, in many more cases, the application is "owned" by the service provider, and cannot be arbitrarily manipulated by the user. Rather, it has a well defined set of ways in which it is invoked and interacted with. The

application operates on behalf of a user. That user might be the one interacting with it (as in the buddy list application), or might be representing the interests of that user when a different user interacts with it (as in the authorization application). In this model, we assume that an application can always determine the user on whose behalf it is operating for any particular interaction.

The application, in order to properly operate, requires a set of data elements that are specific to the user on whose behalf it operates. Each application has a well-defined set of data elements it requires. Each data element is named (for example "buddy list") and is of a well-defined type. Example types include lists, integers, trees, or scripts.

In order to provide those data elements, the user can create, destroy, and manipulate objects of various types. It can also bind an instance of an object to a particular named data element. As an example, a user can create a list called "my friends" and bind it to the "buddy list" data element, which accepts lists.

This model is shown pictorially in Figure 2. In this model, the application has a number data elements (represented by "holes"), each of which has a name (foo, bar, baz). The shape of the hole represents its type. There is a data storage (which could be the Internet itself) that contains data elements of various types. Authorized users can bind the holes to instances of each shape (type), in order to fully define the operation of the application. There is a default object that would fill in each hole when one has not explicitly been provided.

4.2 Examples

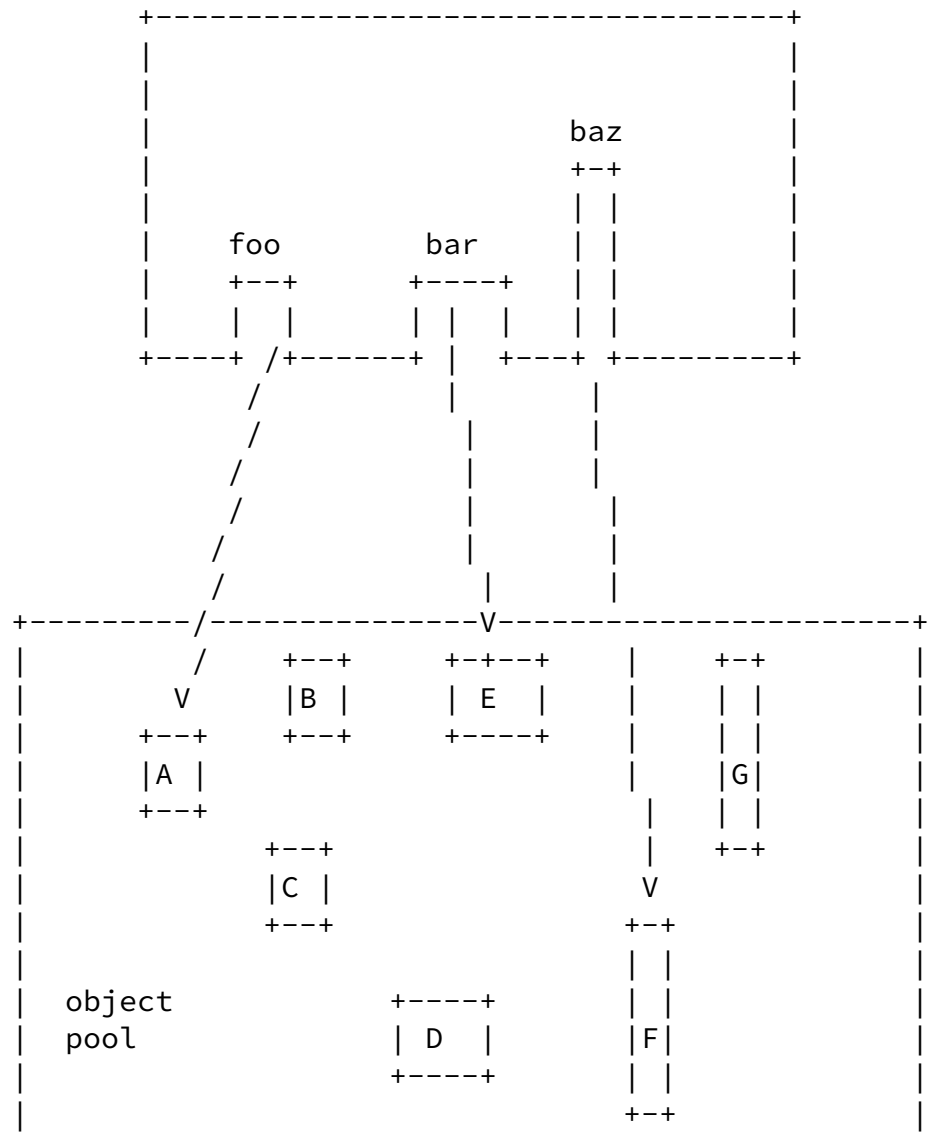
The model is easily applied to many different applications.

A simple example is call-forward no-answer. This application requires a single data element, the call-forwarding number. This data element is a particular data type (phone number). It should be possible for a user to create several phone numbers within the network, associate each with a name, and then bind the actual application to a specific name.

Another example is a voicemail application. The application might

require a number of data elements - the number of rings until it goes to voicemail (an integer), the prompt played to the caller (an audio file), and the password for accessing the voicemail (a string). Clearly, some of these data elements, like the greeting, cannot be manipulated through a protocol proposed here. However, the binding

Example Application



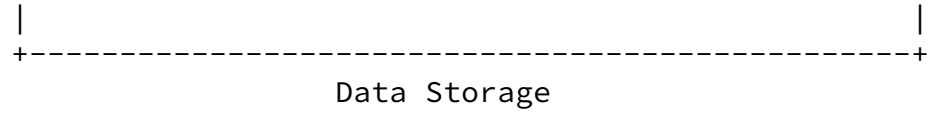


Figure 2: Data Manipulation Model

operations proposed here are applicable. In the example of the voicemail greeting, the greeting could be recorded separately, but assuming it is nothing more than a URI (such as sip:greeting33@example.com) it can be bound to the "greeting" data

element of the voicemail application through the protocol whose requirements are discussed here.

4.3 Requirements

Based on this model, we propose the following general requirements for a protocol to manipulate user feature data:

REQ 1: It MUST be possible to create objects that have one of several defined types. The types MUST include, at a minimum, integer, string, list of strings, list of URI, and blob.

REQ 2: It MUST be possible for the user to provide a URI that identifies the object that was created.

REQ 3: It MUST be possible for the server to provide the client with a URI that identifies the object that was created.

REQ 4: It MUST be possible to destroy an object.

REQ 5: It MUST be possible to query for the value of a particular object.

REQ 6: It MUST be possible to perform type-specific manipulations on the object. In the case of lists, this would include addition and removal of members.

REQ 7: It MUST be possible to bind a named data element for a particular named application to a particular object. For example, a client could bind the buddy list data element of the Presence application to the buddy list myfriends@example.com.

REQ 8: It MUST be possible to obtain the name of the object bound to a particular data element of a particular application. For example, to determine which buddy list is the current one in use.

REQ 9: It MUST be possible for the client to maintain a cached copy of a particular object.

REQ 10: It MUST be possible for multiple clients to maintain a cached copy of the same object.

REQ 10: It MUST be possible for the client to receive notifications of changes to an object.

REQ 11: It MUST be possible for the client to perform type specific manipulations on an object even while not connected to the server.

REQ 12: It MUST be possible for an object to be resynchronized to the master copy on a server, once a client reconnects.

REQ 13: Manipulations of data objects MUST exhibit the ACID property.

REQ 14: It SHOULD be possible for client to learn the set of data elements, and their types, for a particular named application. As an example, a client could query a voicemail application, and learn that it requires an integer called "number of rings" and an audio file called "greeting".

REQ 15: It MUST be possible for the server to authorize only specific users to create, destroy, and manipulate objects, and to bind an object to a data element.

REQ 16: It MUST be possible for the server to authenticate the client, and for the client to authenticate the server.

REQ 17: It MUST be possible for message integrity to be provided for all messages between client and server, and server and client.

REQ 18: It MUST be possible to provide privacy for all messages exchanged between client and server.

5 Possible Solutions

This document is primarily a requirements document, and does not aim to provide a protocol for meeting the requirements defined here. However, there are several protocols already in existence which appear close to meeting the requirements described. One of these is ACAP [6]. Since the protocol is primarily a client-server RPC type of operation, it seems like HTTP and SOAP might also serve as a basis, with a suitably defined set of WSDL. SIP could operate alongside SOAP, to provide the notification aspects of the requirements. SNMP is another possibility for the protocol.

6 Authors Addresses

Jonathan Rosenberg
dynamicsoft

72 Eagle Rock Avenue
First Floor
East Hanover, NJ 07936
email: jdrosen@dynamicsoft.com

Markus Isomaki
Nokia
Nokia House
Keilalahti, Espoo
Finland
email: markus.isomaki@nokia.com

7 Normative References

8 Informative References

[1] M. Day, J. Rosenberg, and H. Sugano, "A model for presence and instant messaging," RFC 2778, Internet Engineering Task Force, Feb. 2000.

[2] J. Rosenberg, "A SIP event package for buddylist presence," Internet Draft, Internet Engineering Task Force, June 2002. Work in progress.

[3] J. Rosenberg, "Session initiation protocol (SIP) extensions for presence," Internet Draft, Internet Engineering Task Force, May 2002. Work in progress.

[4] D. Crocker et al. , "A common profile for instant messaging (CPIM)," Internet Draft, Internet Engineering Task Force, Nov. 2001. Work in progress.

[5] J. Lennox and H. Schulzrinne, "Call processing language framework and requirements," RFC 2824, Internet Engineering Task Force, May 2000.

[6] C. Newman and J. G. Myers, "ACAP -- application configuration access protocol," RFC 2244, Internet Engineering Task Force, Nov. 1997.
