Group Messaging System Software Architecture and API

Amitabh Saxena, Kapali Vishwanathan, Guruprasad Kini

HP Laboratories
HPL-2011-208R1

Keyword(s):
Group-messaging-system; multi-tenant group-messaging system; capability-based addressing

Abstract:
The Group Messaging System (GMS) is a messaging system designed for use in ad-hoc groups. It is a centralized cloud-based service that allows people to communicate using capability based addressing. This document gives an overview of the software architecture of the prototype implementation of GMS.

External Posting Date: March, 9 2012 [Fulltext]
Internal Posting Date: March 9, 2012 [Fulltext]

Approved for External Publication

Group Messaging System Software Architecture and API

Amitabh Saxena, Kapali Vishwanathan, Guruprasad Kini

{amitabh.saxena, kapali, guruprasad.kini}@hp.com

Abstract

The Group Messaging System (GMS) is a messaging system designed for use in ad-hoc groups. It is a centralized cloud-based service that allows people to communicate using **capability based addressing**. This document gives an overview of the software architecture of the prototype implementation of GMS.

1. Introduction

The theory and architecture GMS was presented in an earlier technical report[1]. The basic construct in GMS are cells which are communication abstractions addressable via two 256 bit numbers. Apart from cells, there are four other similar constructs that are addressable via a 256 bit number. These are:

1. Queues: a 256 bit address for queuing and retrieving messages from cells
2. Forwarders: a 256 bit address for sending messages to many cells
3. GID: the group ID, a 256 bit address identifying a group
4. Member-ID: a 256 bit address identifying a public key. This is the fingerprint of the key.

GMS provides three different APIs for using these constructs in applications. These are:

GMS API: The GMS server provides a query based interface for managing cells and various links. The GMS client (GMC) has the appropriate API for making such queries to the GMS server. This is the lowest level of access GMS provides.

Flat groups API: At a slightly higher level, we have developed a pub-sub based messaging system with group semantics. The groups so created are called "flat groups" because all members are considered at the same access level (see Fig 1 to understand why they are called flat groups). The group semantics imply that users no longer deal with individual cells or links, but rather deal with creating groups, adding and removing members from groups, and reading from or writing to groups. Although this middle level API provides us with group semantics, it still deals with addresses (256 bit random numbers), and is not therefore very programmer/user friendly. To overcome this problem, we have a third layer API that takes care of mapping addresses to human understandable names.

[1] HPL-2011-69: Communication cell: Cryptography and capability for communication access control. Saxena, Amitabh; Viswanathan, Kapali.

Multi-tenant (MT) Flat Groups API: This API allows several users to use the flat-group system of GMS. Different users are identified by usernames. Furthermore, it has the following features: group addresses (GIDs) are transparently mapped to human readable group nicknames, Member IDs (Public key fingerprints) are transparently mapped to contact nicknames. The multi-tenant API can be used in both single-tenant (ST) and multi-tenant mode. When using in multi-tenant mode, an additional authenticating feature is required (no authentication is done in single-tenant mode). Currently this authentication is done via a password. However, later versions will add support for more advanced authentication mechanisms. Since ST-mode is a subset of MT mode, we will only discuss MT mode here. This is done in Section 3.

2. Flat Groups in GMS

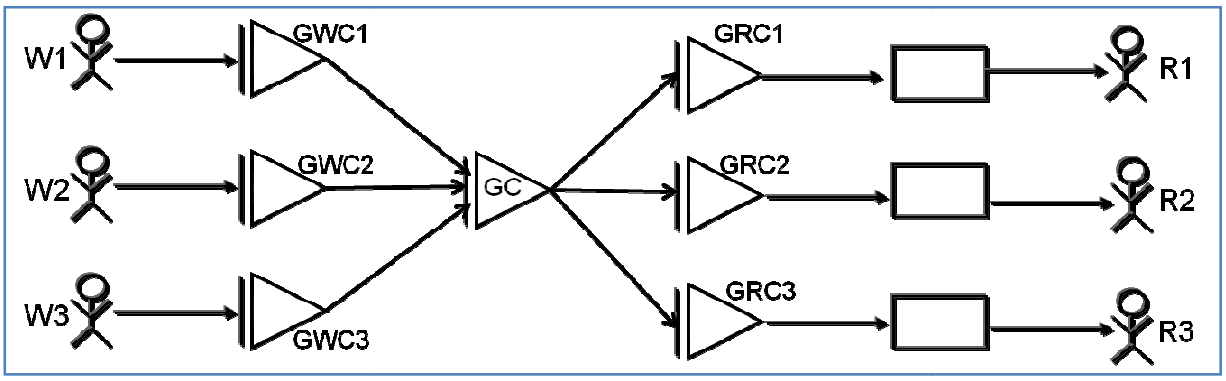

Figure 1: Flat groups in GMS

GMS provides two distinct communication semantics:

1. One-to-one communication (hereafter called “chat”), and
2. Many-to-many communication (hereafter called “groups”)

There are several ways to implement 1 and 2 using cells. In this section, we describe the architecture of one specific example of 2 that we implemented as a prototype. This document explains how our prototype is designed on top of GMS. A similar document will be available to describe an implementation of 1. Since GMS is based on the underlying concept of cells, we first give a brief overview of cells.

Preliminaries: The basic construct of GMS is “cells”. A Cell is a virtual box with two openings, called addresses. One address (the Input address – IA) is used for writing to the cell and the other address (the output address – OA) is used for retrieving messages written to the cell. Generally, given one of the addresses, it is computationally hard to find the other.

As discussed previously, the GMS server consists of three types of cells:

1. RW cells (IA and OA both look pseudorandom)
2. R cells (IA is computed from public key and OA looks pseudorandom)

3. W cells (OA is computed from public key and IA looks pseudorandom)

Note: GMS groups use only the first two types of cells (RW and R).

The GMS server additionally provides the following constructs:

1. Forwarders are used for writing to multiple cells. A forwarder has a single address – the forwarder address or FA.
2. Queues are used for reading from cells. A queue has a single address – the queue address or QA.

GMS allows the following links between its components:

1. A FA can be linked to the IAs of several cells (of any type), so that any message sent to FA is forwarded to all cells whose inputs are linked to it. The links between FAs and IAs are called F-Links
2. A QA can be linked to the OAs of several cells (of any type), so that any message arriving on those cells is retrieved by the queue. The links between QAs and OAs are called Q-Links
3. An RW-cell can forward messages to another RW-cell. That is, messages coming out from the OA of one RW-cell can be sent to the OA of another RW-cell. Such a link between RW-cells is called a C-Link.

GMS provides the following access control for manipulating and viewing links:

1. To make/break F-Links, both the corresponding FA and IA are needed.
2. To make/break Q-Links, both the corresponding QA and OA are needed.
3. To make/break C-Links, both IA and OA of the sending cell, and the IA of the receiving cell are needed.
4. To view F-Links, the corresponding FAs are needed. It is not possible to view F-Links by supplying the IAs
5. To view Q-Links, the corresponding QAs are needed. It is not possible to view Q-Links by supplying the OAs
6. To view C-Links, the corresponding (IA/OA) pairs of the sending RW-cells are needed. It is not possible to view Q-Links by supplying the IAs of receiving cells.

How groups are created and used:

In GMS, the following information is needed to create groups:

1. List of public keys of readers
2. List of public keys of writers
3. List of public keys of managers
4. Public and private keys of creator
5. Group name and description

Let us call the info in 4-5 as auxiliary information, as it is not used by the server in any way (it is used only by the clients).

On receiving the request, the server does the following:

1. First it generates an RW-cell for the group. Call this cell the group cell (GC)
2. For each writer, it generates an RW-cell. Call these cells GWCs (group write cells)
3. For each reader, it generates an RW-cell. Call these cells GRCs (group read cells)
4. For each manager, it generates an RW-cell. Call these cells GMCs (group manage cells)
5. From each GWC, make a C-Link with GC, so that that messages sent to each GWC will be sent to IA of the GC. See Figure 1.
6. Make a C-Link from GC to each GRC, so that messages coming out from GC will be sent to the IA of each GRC. See Figure 1.
7. For each reader, send the OA of the corresponding GRC to the IA of the R-cell of that reader. (Recall that this IA is computed from the public key of the reader). The auxiliary information is also sent. See Figure 2.
8. For each writer, send the IA of the corresponding GWC to the IA of the R-cell of that writer. The auxiliary information is also sent. See Figure 2.
9. For each manager, send the IA of the corresponding GMC to the IA of the R-cell of that manager. The auxiliary information is also sent. See Figure 2.
10. Store the IAs of all GWCs, GRCs and GMCs, and also store the GC.

All potential members (readers/writers/managers) must a priori be listening to the OAs of their corresponding R-cells. After the server sends the messages in Steps 7-9 above, the members receive these “invites” on their R-cells, and based on the auxiliary information, they can decide whether to join the group or not. Join is as follows:

1. Readers will subscribe the OA to their data queue
2. Writer will use IA to write to the group when needed.
3. Managers will use IA to manage the group (edit membership)

Figures 1 and 2 describe how the groups are created and used

The following notation is used in the figures:

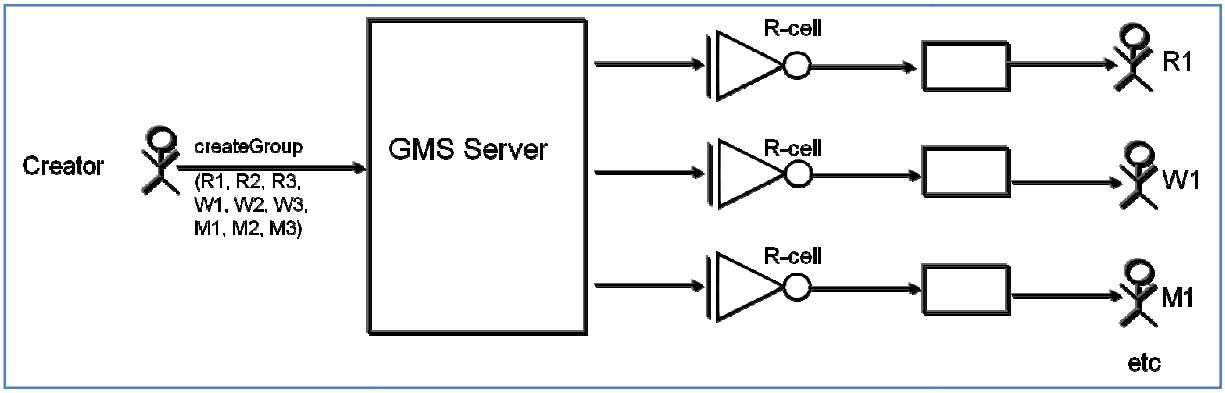

Figure 2: Group creation

3. Multi-tenant Flat Groups API

The multi-tenant (MT) API is the top level API available for the GMS system. Essentially, this API gives access to the flat-group API in a more human-friendly manner, by translating addresses to nicknames. Secondly, it supports multiple users and allows thin clients (JavaScript) to access the services via a browser. A MT API when used in a ST mode can be considered as a thick client. The following is a comprehensive list of the MT-Client API for the JVM. The notation is as follows:

functionName(para1:para1Type, para2:para2Type, ...): functionReturnType

Part 1: Managing/Editing groups

```
addContact(userName:String, contactNick:String, contactUserName:String, auth:String): Status

addContactFromKey(userName:String, contactNick:String, contactPubKey:PublicKey, auth:String): Status

addToGroup(userName:String, groupNick:String, readers:Array[String], writers:Array[String],
managers:Array[String], reason:String, strict:Boolean, auth:String): Status

changePassword(userName:String, oldPass:String, newPass:String): Status

createGroup(userName:String, groupNick:String, groupName:String, readers:Array[String],
writers:Array[String], managers:Array[String], desc:String, strict:Boolean, auth:String): String

deleteContact(userName:String, contactNick:String, auth:String): Status

deregisterUser(userName:String, password:String): Status

destroyGroup(userName:String, groupNick:String, reason:String, auth:String): Status

exportGroup(userName:String, groupNickName:String, auth:String): String

getContacts(userName:String, auth:String): Array[String]

getGroupInfo(userName:String, groupNick:String, auth:String): GroupInfo

getGroupMembers(userName:String, groupNick:String, auth:String): GroupMemberNicks
```

getGroups(userName:String, auth:String): Array[String]

getMyManageGroups(userName:String, auth:String): Array[String]

getMyReadGroups(userName:String, auth:String): Array[String]

getMyWriteGroups(userName:String, auth:String): Array[String]

isAvailable(userName:String): Boolean

isManager(userName:String, groupNick:String, auth:String): Boolean

isMember(userName:String, groupNick:String, auth:String): Boolean

isReader(userName:String, groupNick:String, auth:String): Boolean

isWriter(userName:String, groupNick:String, auth:String): Boolean

joinAsManager(userName:String, groupName:String, contactNick:String, cookie:String, auth:String): String

joinAsReader(userName:String, groupName:String, contactNick:String, cookie:String, auth:String): String

joinAsWriter(userName:String, groupName:String, contactNick:String, cookie:String, auth:String): String

leaveAsManager(userName:String, groupNick:String, auth:String): Status

leaveAsReader(userName:String, groupNick:String, auth:String): Status

leaveAsWriter(userName:String, groupNick:String, auth:String): Status

loadUser(userName:String, password:String): Status

pingGroup(userName:String, groupNick:String, appID:String, auth:String): Status

pingResponse(userName:String, groupNick:String, appID:String, senderInfo:String, auth:String): Status

refreshGroup(userName:String, groupNick:String, auth:String): Status

registerUser(userName:String, password:String): Status

removeFromGroup(userName:String, groupNick:String, readers:Array[String], writers:Array[String], managers:Array[String], reason:String, auth:String): Status

renameContact(userName:String, oldNick:String, newNick:String, auth:String): Status

renameGroup(userName:String, oldNick:String, newNick:String, auth:String): Status

unloadUser(userName:String, password:String): Status

writeToGroup(userName:String, groupNick:String, appID:String, message:String, auth:String): Status

Part 2: Reading from groups

addDataHandler(dh:TraitDataHandler): Status

addControlHandler(ch:TraitControlHandler): Status

removeDataHandler(handlerID:String): Status

removeControlHandler(handlerID:String): Status

getDataHandlers: Array[TraitDataHandler]

getControlHandlers: Array[TraitControlHandler]

getDataHandler(handlerID:String): TraitDataHandler

getControlHandler(handlerID:String): TraitControlHandler

isDataHandlerAttached(handlerID:String): Boolean

isControlHandlerAttached(handlerID:String): Boolean