Communication Middleware for a Web-based Game Lobby

Yanwar Asrigo

Master of Science

School of Computer Science

McGill University
Montréal, Québec
October 2009

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

Copyright © 2009 by Yanwar Asrigo
All rights reserved
ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisors Bettina Kemme, Jörg Kienzle, and Clark Verbrugge for giving me the opportunity to work on this thesis. This work would not have been possible without their ideas, opinions, comments, and financial support. I would also like to thank Quazal, especially Carl Dionne and Alexandre Quesnel, for their invaluable support. This work has benefited greatly from their knowledge, technical expertise, and assistance.

Thanks also goes to the School of Computer Science Help Desk team who worked tirelessly to maintain a stable computing environment for the research at the department. Numerous thanks go to Andrew Bogecho and Kailesh Mussai who have provided great help during critical stages of my experiments.

Finally, I would like to thank my family who encouraged me to apply for graduate study and steadily gave moral support throughout my study.
ABSTRACT

A game lobby is an online service provided to online game players. Players interact with each other and participate in games through the game lobby. As web technology becomes more advanced, the web is seen as a promising platform for online game lobbies.

In this thesis, we develop a web-based communication middleware for game lobbies using the XML-RPC web service framework. Web-based communication middleware must use the Hypertext Transfer Protocol (HTTP) for its communication transport, which is based on the request-reply protocol, and requires all communication to be initiated by the client. This presents a major challenge for realizing the communication middleware as the lobby server needs to send information to the clients at arbitrary time points with minimum latency. We propose a novel Server Push Enabled XML-RPC (SPEX) protocol to address this problem by employing the so-called “long-polling” technique. We also devise a light-weight authentication scheme, called Authenticated XML-RPC. We have implemented SPEX and Authenticated XML-RPC and analyzed the performance characteristics of our implementation. Experiments show that the communication middleware scales well, providing good response time for the kind of real-time interactions that occur typically in an online game lobby setting.
Un lobby de jeu est un service en ligne fourni pour les joueurs en ligne. Ces derniers peuvent interagir entre eux et prendre part aux jeux à partir du lobby de jeu. Grâce aux avancements technologiques, le Web est perçu comme une plateforme prometteuse pour développer des lobbies de jeux en ligne.

Dans ce mémoire de maîtrise, nous développons un intergiciel de communication Web pour des lobbies de jeux en utilisant le cadre de service Web XML-RPC. Tout intergiciel de communication Web doit utiliser l’Hypertext Transfer Protocol (HTTP) comme transport de communication, qui est basé sur un protocole de transmission sur demande et nécessitant que toutes communications soient initiées par le client. Ceci pose un défi important à l’élaboration d’un intergiciel de communication vu que le serveur du lobby doit envoyer des informations aux clients à des temps arbitraires et avec un temps d’attente minimum. Pour aborder ce problème, nous proposons un nouveau protocole Server Push Enabled XML-RPC (SPEX) en exploitant la technique de “long-polling”. Nous avons également conçu un système allégé d’authentification intitulé Authenticated XML-RPC. Nous avons mis en œuvre SPEX et Authenticated XML-RPC, ainsi qu’analysé les performances de notre implémentation sous différentes configurations. Nos expériences ont démontré que notre intergiciel de communication est extensible tout en garantissant un temps de réponse satisfaisant pour ce genre d’interactions en temps réel survenant généralement dans un environnement de lobby de jeu en ligne.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................ ii

ABSTRACT ................................................... iii

ABRÉGÉ ..................................................... iv

LIST OF TABLES ............................................. viii

LIST OF FIGURES ........................................... ix

1 Introduction ............................................. 1

2 Background and Related Work .............................. 5

  2.1 Game Lobby Services .................................. 6
  2.2 Game Lobby Architecture .............................. 8
  2.3 Game Lobby Communication Middleware ............... 10
  2.4 Game Lobby Traffic Characteristics .................. 14
  2.5 HTTP ................................................. 16
  2.6 Enabling Server Push Traffic over HTTP ............. 17
    2.6.1 Comet Technique .................................. 17
    2.6.2 BOSH Technique from XMPP ...................... 18
  2.7 XML-RPC .............................................. 19

3 Authenticated XML-RPC .................................. 23

  3.1 Motivation ........................................... 23
  3.2 Requirements ....................................... 25
  3.3 Design ............................................. 26
  3.4 Login Protocol ...................................... 30
  3.5 Security Threats .................................... 31
    3.5.1 Replay Attack .................................... 31
    3.5.2 Message Spoofing ................................ 32
    3.5.3 Man-in-the-Middle Attack ....................... 32
3.5.4 Security of MD5 ........................................ 33
3.5.5 Security of the Login Protocol ......................... 33
3.5.6 Password Storage at the Server ....................... 34
3.6 Enabling Information Privacy .......................... 35
4 SPEX: Server-Push Enabled XML-RPC ..................... 36
  4.1 Motivation ............................................ 36
  4.2 Design Overview .................................... 37
  4.3 SPEX Framework ..................................... 41
  4.4 SPEX Mono Algorithm ................................ 43
    4.4.1 Connection Setup .................................. 43
    4.4.2 Processing Application RPC Requests and Server Updates 45
  4.5 SPEX Duo Algorithm ................................ 49
    4.5.1 Connection Setup .................................. 49
    4.5.2 Processing Application RPC Requests and Server Updates 50
  4.6 Comparing SPEX Mono and Duo ......................... 61
  4.7 Comparing SPEX, Standard XML-RPC, and XMPP BOSH ... 64
5 Implementation ........................................... 67
  5.1 SPEX Server .......................................... 67
    5.1.1 SPEX Mono ........................................ 68
    5.1.2 SPEX Duo .......................................... 72
    5.1.3 Authenticated XML-RPC Support .................... 74
    5.1.4 Implementation Challenges ....................... 74
  5.2 SPEX Client .......................................... 75
6 Experiments ................................................ 79
  6.1 Traffic Scenario ...................................... 79
  6.2 Methodology ......................................... 80
  6.3 Test Application Implementation ..................... 82
  6.4 Infrastructure ....................................... 83
  6.5 Bandwidth Estimation ................................ 84
  6.6 Experimental Results ................................. 85
    6.6.1 SPEX Server without Thread-Pool ................... 85
    6.6.2 SPEX Server with Thread-Pool ..................... 88
    6.6.3 SPEX Server Using concurrentMcast .................. 90
    6.6.4 Authenticated XML-RPC ............................. 93
6.7 Summary ................................................................. 94

7 Conclusion and Future Works ........................................ 96

References ........................................................................ 98
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3–1</td>
<td>(&lt;qReqInfo&gt;) elements</td>
<td>28</td>
</tr>
<tr>
<td>3–2</td>
<td>(&lt;qSecurity&gt;) elements</td>
<td>29</td>
</tr>
<tr>
<td>6–1</td>
<td>Machine setup for the experiments</td>
<td>83</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2–1</td>
<td>High level architecture of a game lobby</td>
<td>9</td>
</tr>
<tr>
<td>2–2</td>
<td>Client-server interaction</td>
<td>11</td>
</tr>
<tr>
<td>2–3</td>
<td>One-to-one interaction</td>
<td>12</td>
</tr>
<tr>
<td>2–4</td>
<td>One-to-many client interaction</td>
<td>12</td>
</tr>
<tr>
<td>2–5</td>
<td>Server-client interaction</td>
<td>13</td>
</tr>
<tr>
<td>2–6</td>
<td>BOSH infrastructure deployment</td>
<td>19</td>
</tr>
<tr>
<td>2–7</td>
<td>XML-RPC framework</td>
<td>20</td>
</tr>
<tr>
<td>2–8</td>
<td>XML-RPC request for Add procedure</td>
<td>21</td>
</tr>
<tr>
<td>2–9</td>
<td>XML-RPC response for Add procedure</td>
<td>21</td>
</tr>
<tr>
<td>3–1</td>
<td>High-level architecture of XML-RPC communication</td>
<td>26</td>
</tr>
<tr>
<td>3–2</td>
<td>Authenticated XML-RPC request for Add procedure</td>
<td>29</td>
</tr>
<tr>
<td>3–3</td>
<td>Authenticated XML-RPC response for Add procedure</td>
<td>30</td>
</tr>
<tr>
<td>3–4</td>
<td>Authenticated XML-RPC Login Protocol</td>
<td>31</td>
</tr>
<tr>
<td>4–1</td>
<td>Server push mechanism in SPEX</td>
<td>38</td>
</tr>
<tr>
<td>4–2</td>
<td>Server push mechanism in SPEX</td>
<td>42</td>
</tr>
<tr>
<td>4–3</td>
<td>SPEX Mono login protocol</td>
<td>44</td>
</tr>
<tr>
<td>4–4</td>
<td>SPEX Mono client protocol</td>
<td>46</td>
</tr>
<tr>
<td>4–5</td>
<td>SPEX Mono server protocol</td>
<td>48</td>
</tr>
</tbody>
</table>
6–7 SPEX server with a thread-pool of 50 threads RTT measurements . . 91
6–8 SPEX server using concurrentMcast RTT measurements . . . . . . . 92
6–9 concurrentMcast and iterativeMcast performance comparison . . . 93
6–10 Threaded SPEX server with Authenticated XML-RPC RTT measure-
ments ................................................................. 94
CHAPTER 1
Introduction

In the early days of multiplayer online games (MOGs), games were normally played in a private network among friends. As internet connections become more accessible, many more people have become interested in MOGs. People from different parts of the world can now meet up online and play the same game. In order to do so, however, players must search for other players with similar interests, typically through ad-hoc communication channels, such as forums and mailing lists. The challenge does not stop there as organizing a game session proves not to be that simple either. Players have to exchange many emails or messages just to agree on a specific time to play a game, taking time zones into consideration. In addition, a game server has to be setup correctly and strategically so that it is accessible by all players. Doing all of this requires a considerable amount of effort and time compared to the short-lived game session itself.

With the flourishing popularity of online games, there is a critical need for a centralized service to obviate the pre-game hassles. Such a service is known as a game lobby. A game lobby is a virtual place where players can find others and create and join games. A player establishes their profile in a game lobby so that he/she can be discovered by other players. Players can also interact with each other in the game lobby for exchanging tips, planning the next game, etc. When a player initiates a game, the lobby automatically invites the enlisted players and establishes the MOG
session. The game lobby can also provide persistence service, for example for storing match statistics which can serve as players’ track record. In addition, a game lobby may also support multiple game titles.

Today, most game lobbies are deployed as a client-server application. The server provides the back-end services which perform the lobby functionalities. The client application provides the user interface to access the back-end services and to render information or notifications pushed by the server. Thus, communication between the client and the server runs in both directions. Moreover, some of the lobby traffic is time sensitive to support real-time interactions, such as instant messaging or coordination messages from the server to start a game.

Lately, with the rapid advances of web technology and proliferation of broadband networks, the web is able to deliver rich contents to end users on demand. Many applications, including game lobbies, are seeing the web as an appealing platform for deployment. Furthermore, web traffic is rarely blocked in the internet, which makes web applications easily accessible. Web-based applications normally use a constrained thin client, that is a web-browser, to communicate with the server. At the server side, the hosted application functionalities are normally exposed to the client through web services to promote interoperability. The client application makes a web service request to initiate a computation to be performed by the server and then receives a response that carries the execution result. Since web service technology is an extension of web technology, the communication uses Hypertext Transfer Protocol (HTTP) for transport with the exchanged information encoded in Extensible Markup Language (XML). Moreover, the web service technology is also commonly adopted
by non web-browser based applications for their communication because of its cross
network accessibility and interoperability.

There are many challenges in developing a web-based game lobby. However, in
this thesis, we only focus on the challenges in implementing a communication
middleware for the web-based game lobbies. The client-server paradigm of web-
based applications suits the game lobby’s client-server architecture. The web-based
game lobby is restricted to HTTP for communication transport. HTTP is a request-
response protocol. All communications start from the client by sending a request to
the server. Only after receiving a client request can the server send a response back
to the client. Unfortunately, HTTP does not allow the server to send data to the
client at its discretion. However, as we commented earlier, the game lobby server
also needs to push information to clients in real-time at arbitrary time points. Thus,
the main question that arises is how to enable the server to push traffic over HTTP
for web-based game lobby. In this thesis, we design a game lobby communication
middleware using the XML-RPC [3] web service framework that allows the server to
send messages to the client.

Contributions

More precisely, the two main contributions of this thesis are:

- *Authenticated XML-RPC scheme*. There is no standard available for securing
  XML-RPC messages. We design a lightweight security scheme that is suitable
  for game lobby traffic to ensure the authenticity of XML-RPC communication.
• **Server-Push Enabled XML-RPC (SPEX).** We extend the standard XML-RPC to enable emulated bidirectional communication by using the so-called “long-polling” technique.

• **Game lobby communication middleware.** We implement a web-based communication middleware suitable for game lobbies using Authenticated XML-RPC and SPEX and evaluate its performance.

**Roadmap**

The remainder of this thesis is structured as follows: the next chapter presents background information on game lobbies and internet communication and also related works on enabling bidirectional-streams over HTTP. Chapter 3 describes the design of our Authenticated XML-RPC scheme. In the following chapter, we discuss the design of SPEX. In Chapter 5, we explain how we implement SPEX together with the Authenticated XML-RPC scheme. Chapter 6 provides the performance measurements taken from our implementation. Finally, we draw some conclusions and discuss future works in the last chapter.
CHAPTER 2
Background and Related Work

A game lobby is a virtual confluence point where the online game community can interact with each other before the game. In Section 2.1, we elaborate the services that a game lobby provides to enhance the online gaming experience. We then discuss the high level architecture and communication models of a game lobby. We also study the traffic characteristics of a game lobby.

With the maturity of web technology, the web can be an excellent platform for game lobby implementations. Players can easily access the game lobby by using a web-browser without having to install anything. Alternatively, the traditional client-server game lobby implementations may use the web as the communication transport to gain the favorable features of web based communication. In any case, they inevitably rely on web services with HTTP for the communication transport. In Section 2.5, we provide some background information on HTTP and the primary problem that HTTP poses for a web-based game lobby implementation. Following that, we discuss two techniques that may help to address the problem. Finally, we describe the XML-RPC web service framework that we used for developing our communication middleware.

In our discussion, we are using the term MOG and game-title interchangeably to refer to a game published by a game publisher. We also use the term game and game-session to refer to a game session being played by multiple players.
2.1 Game Lobby Services

A game lobby offers several functionalities to players playing a MOG. We analyzed the functionalities of Quazal Rendez-Vous®, a full-featured commercialized game lobby implementation, and categorized its functionalities into eight main services as described below.

**Account Management.** Account management enables players to create their profiles. Players publish their profiles so that they can be discovered by others. Different players may assume different roles and each role has its own access rights, for instance, administrators can organize a tournament.

**Friends.** A game lobby may capture the social network of the players as well. Players create player-lists consisting of the people that they usually interacts with. Players can classify their relationships with other players, for example, friend, adversary, or blacklisted. The friends service may also report updates from other players, such as when a friend logs in or wins a combat.

**Messaging.** The messaging service enables players to send messages to others. Messages can be in the form of one-to-one, like chat and email, or one-to-many, like group chats. The messaging service may allow a player in the lobby to send a message to another player in a game. The game lobby may also be configured to relay messages from one game to another game, even if the games belong to different titles.

**Matchmaking.** One fundamental functionality of a game lobby is to enable players to create, find, and participate in a game session. An authorized player can create a game session and enlist his friends to join. Alternatively, he can open the
game to the public by specifying some criteria. At the set time, the matchmaking service notifies the participants that the game is starting and kick-starts the session management service to create the game session.

Session Management. The session management service takes care of setting up and managing game sessions. It interfaces with the MOG and the clients when launching a game session. After the game is completed, the session manager collects the statistics of the game. In addition, the session manager can also retrieve the game statistics while the game is playing and publish it in the lobby.

Persistent Store. The persistent store adds an extra dimension to MOGs, that is continuity between games. The persistent store records statistics and outcomes across the, otherwise, short-lived game sessions. Using the historical data, a reputation or experience system can be established. The historical information is maintained even after the player signs out of the lobby.

Competition. The competition service manages the organization and outcome of matches between players and maintains statistical values concerning any given matches. The service automatically creates the necessary matchmaking for the matches.

NAT Traversal. Many games require peer-to-peer support. In addition, even some lobby services can be set to bypass the lobby, for example two players can chat with each other directly without routing the messages through the lobby server. Since player machines are typically behind firewalls and network-address-translators (NAT), establishing a direct connection between two players may not be feasible.
The game lobby which is directly connected with the players can help to negotiate with the NAT devices to establish direct connections between players.

2.2 Game Lobby Architecture

The online game lobby typically adopts a client-server architecture. The client-server architecture is favored because it gives more control to the lobby administrator. Considering the size of the user base, the centralized approach makes it easier to maintain the consistency of the lobby state. In addition, server upgrades can be easily deployed while client upgrades can be pushed automatically. With the client-server paradigm, the server hosts the lobby services while the client interacts with the server to access those services. The client is also responsible in presenting information received from the server and capturing user activities.

Figure 2–1 depicts a high level architecture of a generalized game lobby which is derived based on our analysis of the commercial lobby Rendez-Vous sold by Quazal. The server is comprised of four major modules: (1) the Lobby Services, (2) Plug-in Support, (3) Communication Middleware, and (4) Administration. The lobby services module provides the implementation of the lobby functionalities. These functionalities are grouped into several main services as we discussed in the previous section. The plug-in support is critical to a game lobby to provide customizability. Lobby administrators can add new game-titles or new services to retain players’ interest. A good plug-in support provides an uninvasive way for adding new features without modifying the already stable lobby implementation. For example, Quazal Rendez-Vous provides a combination of application hooks, Python extension support, and a data-definition-language (DDL) for customizing the lobby. The communication
middleware hides the low level communication complexity from the application by providing communication services. The middleware module is discussed in more details in the next section. The administration module provides a facility to lobby administrators for managing their lobby. Since the lobby is a long running service, automated system health monitoring is important for ensuring the system availability. Lastly, game lobbies also normally use a database to store the persistent data.

![Game Lobby Server](image)

**Figure 2–1:** High level architecture of a game lobby

The client side can be seen as a three layer application. The bottom layer is the *lobby client library*. This library provides the interfaces (APIs) for the client application to interact with the lobby server. The middle layer, the *Application*,
encodes the logic implemented by the lobby developer. Finally, the Presentation layer handles the interaction with the real player. The presentation layer can be as simple as a text-based console or as sophisticated as a complete 3-D environment [18].

While an MOG allows tens of players to play simultaneously, the game lobby has to serve thousands or millions of active players at one time. In order to scale up, the game lobby server can be deployed on a cluster of servers. However, this thesis only addresses the communication between the lobby client and the lobby server. The server deployment configuration should be transparent to the client. Thus, we use the client-server term to refer to a single server or a cluster of servers.

2.3 Game Lobby Communication Middleware

To present to the players the illusion that they are sharing the same virtual space, all clients must have the same view of the game lobby state. Since the state of the lobby constantly changes because of player activities, clients must interact with the lobby server, which has the entire view of the lobby state, to synchronize their states. In addition, the clients also interact with each other as part of the game lobby social dynamics. There are four types of interactions that normally occur in a game lobby: (1) client-server interaction, (2) one-to-one client interaction, (3) one-to-many client interaction, and (4) server-client interaction. Since it is a client-server architecture, all these interactions involve the server. We discuss each type of interaction below.

**Client-Server interaction.** In this interaction, the client sends a request to the server to do an operation, and then the server returns a response message which contains the result of the operation. For example, David, a player in the lobby, wants
to buy 1000 game credits for his lobby account. Using the lobby client, he opens
his account page and purchases the credits. To process this, the client application
sends a request to the Account.PurchaseCredit service at the server to execute the
transaction. The server verifies the payment information and approves the purchase.
The server then sends a response message to the client to notify him of the successful
transaction. This example is illustrated in Figure 2–2.

Figure 2–2: Client-server interaction

One-to-One Client interaction. Many game lobby activities involve inter-
actions between players (called client-client interactions). One possible interaction is
an interaction involving two clients, that is one-to-one client interaction. An example
of a one-to-one client interaction is presented in Figure 2–3. After David purchased
his game credits, he is ready to play his favorite pong game with Kyle, his pong
buddy in the lobby. He searches for Kyle’s name in his friends list to invite him to
a game at 9 PM. This triggers the lobby client to send a request to the Matchmak-
ing.InitiateGame service. The server matchmaking service automatically sends an
invitation to Kyle and also returns an acknowledgement to David to indicate that
the request has been processed. We refer to all communication flowing to the client
initiated by the server as update traffic: the invitation is sent to the client as an
update event consisting of the invitation information.
One-to-Many Client interaction. One-to-many client interaction is another type of client-client interaction. In this interaction, the client accesses a server service that triggers the server to send a message to multiple clients. Such a transmission operation is called a *multicast*. An example of such an interaction is group chat. Continuing with the previous example, David is now also chatting with four of his friends. He wants to know about the new features in the latest release of the pong game. When he clicks the send button to post his message, the client application initiates the *Messaging.Post* service at the server. Upon receiving the request, the server multicasts the message to the clients of his four friends and also sends a response to David to indicate that the message has been posted. This example is depicted in Figure 2–4.

Server-Client interaction. From time to time, the server also needs to send updates to the clients. For example, let’s return to David’s pong game and say Kyle accepted David’s invitation. At 9 PM, the lobby server automatically sends an alert
to David and Kyle to remind them about their game. These alerts are sent to the players’ client as update messages. This example is depicted in Figure 2–5.

![Figure 2–5: Server-client interaction](image)

To support the above interactions, the communication middleware has to provide three types of communication service: (1) *request-response* service, (2) *server send* service, and (3) *server multicast* service. The request-response service is used by the client to invoke a service at the server. The client sends a request to the server to perform an operation, and the server returns the result in the response. The last two services are provided to the server for disseminating updates to the clients. The send service is used by the server to send an update to a client. The multicast service, on the other hand, pushes an update to multiple clients, which are determined by the list of intended receivers provided by the server. However, multicast is not supported in the internet. In practice, the middleware normally resorts to multiple *unicasts* to send the updates, that is employing the send service multiple times.

In addition to providing the communication services, the communication middleware is also responsible for managing the communication links between the client and the server. When a player wants to log into the lobby, the client communication middleware establishes a communication link with the server. The server middleware then accepts the connection and notifies the backend lobby service to handle the client login. Throughout the lobby session, the client and server middleware
also monitor if the communication link is still up, for example, by exchanging ping messages.

2.4 Game Lobby Traffic Characteristics

In designing the game lobby communication middleware, it is necessary to understand the traffic characteristics of a game lobby. Unfortunately, we could not find any literature pertaining to this topic. However, we suppose that the traffic characteristics of a game lobby should be lower-bounded by those of Instant Messaging (IM) and upper-bounded by those of Massively Multiplayer Online Games (MMOGs). Comparing IM and game lobby services, we can see that almost all IM services are present in the game lobby.

An MMOG is a multi-player online game in a massive scale. It is a vast virtual world where thousands, or even millions, of people can concurrently collaborate and interact with each other by following a game play. However, the game pace of MMOGs is normally slower compared to that of MOGs. MMOGs are usually played for a long period of time, spanning several months or years, during which players accumulate skills or experiences as they play. Similar to game lobbies, an MMOG also provides state persistency. In summary, game lobby services can be viewed as a sub-set of MMOG services. The difference is mainly in the services related to the game play in MMOG. Although, more elaborate game lobbies may also have tournaments, plots, or 3-D environments, in which case they are very close to MMOGs. Like game lobbies, IM services and MMOGs also normally use the client-server architecture and have millions of user base.
The traffic characteristics of two popular IMs, *AOL Instant Messenger (AIM)* and *MSN Live Messenger (MSN)*, are reported in [31]. They discovered that the client incoming traffic is much greater than its outgoing traffic. The *presence* and chat traffic consumes most of the bandwidth. However, in terms of number of messages, presence messages contribute the most number of messages, followed by *hints* and chat messages. Chat messages are typically small: almost all messages have less than 100 bytes text length. Furthermore, 90% of MSN chat messages are less than 50 bytes. Presence messages refer to messages informing the change of status of an IM buddy in the buddy list. Hints messages are messages generated automatically by the IM client, for example, a message indicating a user is typing a message.

MMOGs traffic also exhibits similar characteristics as IMs traffic. MMOGs traffic is a bidirectional traffic consisting of tiny packets which are sent at a low rate [7, 8]. This kind of traffic is referred to as *thin streams* [17]. The following statistics are obtained from [7, 8] which is based on *ShenZhou Online* [2], a TCP-based, commercial, mid-sized MMORPG. However, similar properties are also reported in [17] which analyzed traffic traces from Funcom’s *Anarchy Online* [15]. The client outbound packet inter-arrival time varies from 0 ms to 600 ms. The slow rate is because the traffic is generated as a result of player’s activity. Meanwhile, the client inbound packet inter-arrival times are more regular with approximately half of the inter-arrival times being around 200 ms. Furthermore, 98% of client originated packets only have 32 bytes payload, whereas the payloads of server originated packets average around 114 bytes.
By looking at the IM and MMOG traffic characteristics, we can sensibly estimate that the game lobby traffic also comprises of tiny bidirectional streams with small packet size and low packet rate. In addition, Quazal reported that, based on several deployments of their game lobbies, a lobby client sends 5 messages per minute on average. Furthermore, the message distribution is not uniform, with more messages being communicated before the start of a game and after the end of a game.

2.5 HTTP

Hypertext Transfer Protocol (HTTP) [6, 13] is the communication protocol for the World Wide Web. It is a request-reply protocol that is used by web-clients to communicate with web-servers. All communication is initiated by the client. The HTTP request message supports a fixed set of methods (GET, PUT, POST, etc.) to indicate to the web-server on how the payload content should be processed. The HTTP response message reports on the status of the request. The HTTP message format is designed to be flexible enough to accommodate many types of different media, such as HTML, text, and images. Most HTTP implementations use TCP as an underlying transport protocol, although the standard allows any protocol as long as it provides reliable transport. Moreover, port number 80 has been specifically assigned for HTTP traffic. Due to the popularity of HTTP traffic, HTTP traffic on port 80 is rarely blocked by firewalls.

In its first standard version (HTTP/1.0), a HTTP connection is closed immediately after a single request/response pair. This causes a delay in the protocol, because every communication requires a new TCP connection to be set up. An additional lag is incurred due to the TCP slow-start algorithm, which forces every
new TCP connection to transmit data at a slow rate before it can ramp up to the maximum rate it can achieve. Version 1.1 of HTTP (HTTP/1.1) makes several improvements to boost the protocol’s bandwidth performance. HTTP/1.1 provides a persistent connection so that multiple request/response pairs can be sent over a single connection. There are two versions of persistent connections: with and without pipelining. Pipelining lets a client send multiple HTTP requests without having to wait first for the reception of the response to a previous request. The pipelining feature is not commonly used in practice, because many internet proxy servers do not support it yet.

2.6 Enabling Server Push Traffic over HTTP

Many interactive web-applications require the web-server to push data to a web-client at any time with minimum latency. However, the HTTP request-reply communication model does not allow this. The web-server can only transfer data to a client when the client explicitly requests it. There are two popular techniques that can be used to overcome this constraint, that is enabling server initiated transmission: the Comet technique and the BOSH technique from XMPP. Each technique is discussed in the following sub-sections.

2.6.1 Comet Technique

The Comet technique [27] sidesteps the HTTP constraint by using the HTTP/1.1 chunked transfer encoding mechanism. This mechanism is used by a server to complete a request for contents that are dynamically generated and require a long time to complete. The server sends small chunks of data as they are dynamically created. In the comet technique, the client initiates an HTTP request over a persistent
connection. The server then responds to this request using the chunked transfer encoding and never terminates the response. This way, the server has an open line of communication with which it can push available data to the client immediately as data chunks.

2.6.2 BOSH Technique from XMPP

The Bidirectional-Streams over Synchronous HTTP (BOSH) [23] extension from Extensible Messaging and Presence Protocol (XMPP) [28] is another alternative that enables a HTTP server to push data to clients at the server’s discretion. XMPP is a protocol that was originally aimed for near real-time and extensible instant-messaging and presence information. This protocol is currently expanded to be an open technology for near real-time streaming communication by exchanging XML elements (called XML stanzas).

To support web-client chat, XMPP introduces the BOSH technique to emulate a bidirectional-stream over HTTP by using multiple synchronous HTTP request/response pairs. The BOSH technique is essentially a “long-polling” technique. The server is allowed to hold a client request if there is no data available for the client and only send a complete response once the data become available. We will discuss this technique more in detail in Chapter 4.

The BOSH extension is not an integral part of XMPP even in its deployment. XMPP utilized a specialized Connection Manager (CM) that mediates the communication between an XMPP server and the web-clients. The Connection Manager handles the HTTP connections with the web-client and relays the communication
to the XMPP server using XMPP protocol. Figure 2–6 depicts the infrastructure deployment to support BOSH.

![XMPP Server](XMPP data stream)

**HTTP CM**

![HTTP wrapped XMPP data stream](Web Client)

Figure 2–6: BOSH infrastructure deployment

### 2.7 XML-RPC

XML-RPC is one of the most widely used web services. XML-RPC is a remote procedure call (RPC) framework that uses XML to serialize the communication and HTTP as the transport. An RPC framework enables a client program to call a procedure or method in another program running on the server. Figure 2–7 depicts an RPC execution in the XML-RPC framework. The client application specifies the method name together with the parameters that it wishes to be executed at the server to the XML-RPC client library. In this case, we say that the client application makes an RPC request. The RPC request is marshalled into XML format by the XML-RPC client and sent to the server as the payload of an HTTP POST request. At the server side, the XML-RPC server unmarshalls the XML encoded RPC request and dispatches it to the server application to be executed. The return value of the
method execution, called RPC response, is marshalled by the XML-RPC server and returned to the client using the HTTP response message. The reverse process is performed by the XML-RPC client to return the result to the client application. The RPC framework fits the HTTP request-response protocol well. In Figure 2–8, we show an HTTP POST request message carrying an XML encoded RPC request. The response message is shown in Figure 2–9. In this instance, the client tries to invoke the Add procedure at the server which performs addition of two numbers with double precision.

The verbosity of XML means XML-RPC messages are longer than those using binary representation. Nevertheless, XML is preferred due to its interoperability and readability which in turn makes it easier for debugging purposes. Owing to its simplicity, an XML-RPC implementation, particularly for the client side, can be easily found in many frameworks and programming languages.

Figure 2–7: XML-RPC framework
<table>
<thead>
<tr>
<th>Section</th>
<th>L#</th>
<th>XML-RPC Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Header</td>
<td>01</td>
<td>POST /RPC2 HTTP 1.0</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>User-Agent: Apache XML-RPC 1.0</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>Host: 192.168.124.2</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>Content-Type: text/xml</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td>Content-Length: 253</td>
</tr>
<tr>
<td>XML Data</td>
<td>06</td>
<td>&lt;xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot;?&gt;</td>
</tr>
<tr>
<td></td>
<td>07</td>
<td>&lt;methodCall&gt;</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>&lt;methodName&gt;Add&lt;/methodName&gt;</td>
</tr>
<tr>
<td></td>
<td>09</td>
<td>&lt;params&gt;</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>&lt;param&gt;&lt;value&gt;&lt;double&gt;2.41&lt;/double&gt;&lt;/value&gt;&lt;/param&gt;</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>&lt;param&gt;&lt;value&gt;&lt;double&gt;8.95&lt;/double&gt;&lt;/value&gt;&lt;/param&gt;</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>&lt;/params&gt;</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>&lt;/methodCall&gt;</td>
</tr>
</tbody>
</table>

Figure 2–8: XML-RPC request for *Add* procedure

<table>
<thead>
<tr>
<th>Section</th>
<th>L#</th>
<th>XML-RPC Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Header</td>
<td>01</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>Server: Apache/1.3.12 (Unix)</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>Connection: close</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>Content-Type: text/xml</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td>Content-Length: 124</td>
</tr>
<tr>
<td>XML Data</td>
<td>06</td>
<td>&lt;xml version=&quot;1.0&quot;?&gt;</td>
</tr>
<tr>
<td></td>
<td>07</td>
<td>&lt;methodResponse&gt;</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>&lt;params&gt;</td>
</tr>
<tr>
<td></td>
<td>09</td>
<td>&lt;param&gt;&lt;value&gt;&lt;double&gt;11.36&lt;/double&gt;&lt;/value&gt;&lt;/param&gt;</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>&lt;/params&gt;</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>&lt;/methodResponse&gt;</td>
</tr>
</tbody>
</table>

Figure 2–9: XML-RPC response for *Add* procedure
XML-RPC protocol does not provide much reliability in the presence of failures. It only guarantees *Maybe* invocation semantics [10], that is the protocol is not equipped with any fault tolerance mechanism. The RPC response acts as an acknowledgement to the request. However, the client application cannot know if a request was executed on the server when no RPC response is received. The XML-RPC specification does not have any standard recovery protocol, for example, message retransmission when a failure occurs. XML-RPC relies solely on TCP reliability guarantee which also does not cover all types of communication issues. For example, when a TCP communication breaks down because no acknowledgement package is received, the sender has no clue if the receiver has received the package.

We choose XML-RPC for our prototype because of its simplicity and widely available implementation. Although it is simple, XML-RPC is expressive enough to support most operations needed for the game lobby purposes. The RPC framework also fits well with the general procedural programming paradigm.
CHAPTER 3
Authenticated XML-RPC

As much fun as a game has to offer, there will always be some players who find even more fun in exploiting bugs or back doors of the game to cheat to their advantage. The extent of the effort that goes to breaking a game is astounding at times. Unfortunately, the human readable XML format used in XML-RPC communication sets no barrier at all to cheating at the network level. In this chapter, we present a lightweight protocol that can be used to secure XML-RPC messages for online game lobby purposes.

3.1 Motivation

The XML-RPC framework does not have any standard mechanism for securing its traffic. It relies on the security model of the underlying HTTP instead. The HTTP/1.0 provides the Basic Access Authentication scheme. This rudimentary scheme requires a username and password pair, encoded using the base64 encoding, to be sent with every request. The base64 encoding is considered very insecure for representing user credentials as it is extremely trivial to decode. At the same time, there is no way for the client to check the server’s identity.

HTTP Digest Access Authentication is a more elaborate scheme for negotiating credentials without ever sending the password. The scheme utilizes a simple challenge-response paradigm. The server generates a nonce, a random number that is used once, as a challenge for the client. A valid client must be able to compute
the checksum of the username, the password, the given nonce value, and the HTTP header information. The authenticity of the payload is not guaranteed since the HTTP payload is not included in the checksum. The checksum is computed by using the Message-Digest algorithm 5 (MD5) [26] hash function which produces a 128-bit hash value. MD5 is one of the most popular hash functions. One important feature of MD5 is that it is a fixed hash function that takes no key.

HTTP Secure (HTTPS) [25] is the de-facto standard for secure communication in the internet. HTTPS refers to HTTP protocol that is sent over an encrypted Secure Sockets Layer (SSL) connection. The SSL protocol is the basis of the standardized Transport Layer Security (TLS) [12]. An SSL channel provides communication confidentiality, authenticity, and integrity by using a combination of public key and private key cryptography. The public key cryptography is used during the SSL handshake to authenticate the server to the client and to negotiate the private key to be used during the session. However, the cryptography computation degrades the server performance markedly. [19] reports that SSL requires 5-7 times more computational power than standard HTTP for processing transactions, with, the key exchange operation is the most computationally expensive [9]. An empirical study showed that the key exchange operation took around 175 ms on a web-server with a 1.4 GHz Xeon machine [29]. HTTPS computation demand also lowers the number of concurrent connections that can be served by a uniprocessor server, as much as 90% [29].

Employing HTTPS for XML-RPC traffic may not give an acceptable performance for game lobby purposes. XML-RPC implementations predominantly use
HTTP/1.0 which terminates the connection after a single request/response pair. This makes the communication highly inefficient, adding the cost of establishing an HTTP connection to every communication. As a result, HTTPS is less desirable for our communication middleware. Nonetheless, HTTPS overhead is still tolerable for XML-RPC traffic if a persistent HTTP connection is used.

3.2 Requirements

The first step of using XML-RPC for game lobby communication is to enhance the protocol with a security scheme that minimally impacts the communication performance. We first need to identify the security requirements of online game lobby traffic before designing an appropriate scheme. A closer look at game lobby traffic reveals that a major proportion of the messages are multicasted to multiple users. This implies that most game traffic is not very sensitive and ensuring information privacy is not necessary. However, it is still crucial to ensure that messages are not tampered with while in transit and the message sender is verifiable. With this in mind, we only seek to provide authenticated XML-RPC communication. Privacy is still absolutely necessary for some messages and such traffic should be sent over an HTTPS connection. Moreover, the designed protocol must be able to withstand attacks such as replay attacks, man-in-the-middle attacks, and message spoofing. We will discuss these attacks in more details in Section 3.5.

While the design of this Authenticated XML-RPC scheme is based on the generic game lobby scenario, we realize that different game lobbies may call for additional security measures. Therefore, protocol extensibility is an important feature, too. Last
but not least, the scheme must perform efficiently for both HTTP/1.0 and HTTP/1.1 connections.

3.3 Design

Figure 3–1 depicts a high-level view of a message exchange using XML-RPC. Most XML-RPC implementations are built on top of pre-existing HTTP libraries. The XML-RPC layer receives data structures from the application layer and then marshalls them to XML format before passing it to the HTTP library. The reverse operation is performed when a message is received. Considering this architecture, we decided to target our scheme to be applied at the XML data output of the XML-RPC library. This avoids modifying the HTTP library and at the same time makes the scheme appear transparent to the application level. We do not want to handle the platform and language dependent application data as it will render the designed scheme not portable.

![High-level architecture of XML-RPC communication](image)

Our Authenticated XML-RPC is inspired by the HTTP Digest Access Authentication mechanism and Quazal Rendez-Vous communication security design. We favor the HTTP Digest Access Authentication principal of never sending the user
password over the network. We also employ the MD5 hash function as the building block of our scheme.

The scheme tags every XML-RPC message with a message authentication code (MAC). The MAC tag is generated by computing the MD5 hash value of the XML document together with an agreed secret token (webToken). This token is privately known only to the client and the server. The webToken acts as a secret key in this operation because MD5 is an unkeyed function. The token is generated from the user password as outlined in the following formula:

\[
\text{webPassword} = \text{MD5(user\_password || password\_string)}
\]

\[
\text{webToken} = \text{MD5(webPassword || token\_string)}
\]

The || denotes a concatenation operator. The user\_password is the plain text password supplied by the user. The password\_string and token\_string are some predetermined arbitrary strings that are publicly known to the server and the client. The webPassword is used so that the server does not need to keep user passwords in plain text; meanwhile it is easy for the client side to compute the webPassword once a user enters his/her password. The password\_string also serves a security purpose. Many web applications store user passwords by simply taking the MD5 hash of the passwords. Users also have the tendency of using identical password for many applications. If we derive webPassword in the same way, the adversary can obtain the webPassword from other web applications that can be infiltrated easily.

Since a webToken is user specific, the client needs to include the user information in the RPC request so that the server can retrieve the correct user token for verification. Following the Quazal Rendez-Vous scheme, a user is identified using a
principal identifier (pid). The server also assigns a connection identifier (cid) to a user once he has successfully logged into the game. Note that all client connections that originate from a single client machine and belong to the same pid share the same cid. These identifiers are embedded in the RPC request/response by appending them to the XML encoded RPC request/response in XML format as well. We define an XML structure `<qReqInfo>` for this purpose. In addition to the identifiers, we also include additional request or response related information in this structure: request identifier (rid) and response count (rc). The request identifier is used by the client and the server to track client requests. It is generated by the client and is unique throughout a login session. The server then includes the same rid value in the corresponding response. The response count value informs the client about the number of RPC response messages that the server has sent. Table 3–1 lists the elements of `<qReqInfo>`.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Request</th>
<th>Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;pid&gt;</code></td>
<td>Integer</td>
<td>Mandatory</td>
<td>Optional</td>
<td>Principal Identifier</td>
</tr>
<tr>
<td><code>&lt;cid&gt;</code></td>
<td>Integer</td>
<td>Mandatory</td>
<td>Optional</td>
<td>Connection Identifier</td>
</tr>
<tr>
<td><code>&lt;rid&gt;</code></td>
<td>String</td>
<td>Mandatory</td>
<td>Mandatory</td>
<td>Request Identifier</td>
</tr>
<tr>
<td><code>&lt;rc&gt;</code></td>
<td>Integer</td>
<td>None</td>
<td>Optional</td>
<td>Response Count</td>
</tr>
</tbody>
</table>

Precisely, the MAC tag is generated as follows:

$$\text{MAC} = \text{MD5}(\text{XML Data} \ || \ <\text{qReqInfo}> \ \text{XML} \ || \ \text{webToken})$$

As opposed to using webPassword, the MAC is computed using webToken so that if the scheme is breached, for example, the value of webToken can be inferred in some way, the webPassword is not revealed. The MAC tag for the server RPC response
is computed using the corresponding user’s webToken as well so that the client can verify the authenticity of the server. Another important feature is that the webToken is never exchanged between the client and the server. The MAC tag is included in the XML-RPC messages as an element of <qSecurity> XML structure (see Table 3–2). The <qSecurity> structure is appended after <qReqInfo> in the message body. Figure 3–2 and Figure 3–3 show the XML-RPC request and response of the example Add web service in Section 2.7 when Authenticated XML-RPC scheme is used.

Table 3–2: <qSecurity> elements

<table>
<thead>
<tr>
<th>Tag</th>
<th>Type</th>
<th>Request</th>
<th>Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mac&gt;</td>
<td>String</td>
<td>Mandatory</td>
<td>Mandatory</td>
<td>Message Authentication Code expressed as a 32 digit hexadecimal string</td>
</tr>
</tbody>
</table>

Figure 3–2: Authenticated XML-RPC request for Add procedure

Our main motivation for encoding the request information and the security information in XML structures is to support extensibility. The XML format allows
game developers to easily add extra security or other elements to the structure as needed.

3.4 Login Protocol

The Authenticated XML-RPC scheme requires the principal identifier and connection identifier information as part of the security parameter. These identifiers must be negotiated between the client and the server at the beginning of every communication session. In the online game context, this is marked by the user login. Consequently, we also include a login protocol in our design. Our login protocol is a simplified version of that used in Rendez-Vous. The protocol consists of two steps as illustrated in Figure 3–4:

1. getPID. As a first step, the client calls the getPID RPC with its username as parameter to obtain his pid and a login-session. Since the client does not know his pid yet, the getPID request and response are signed using the webToken default value.
2. connect. Next, the client can request a connection from the server by calling the connect procedure to obtain a cid. The client needs to supply the pid and the login-session returned by the getPID procedure. In addition, the client also includes a nonce in the parameter as a challenge for the server. A valid server will be able to return a cid and the nonce tagged with a valid MAC tag.

![Authentic XML-RPC Login Protocol](image)

Figure 3–4: Authenticated XML-RPC Login Protocol

3.5 Security Threats

In the following sub-sections, we discuss how the Authenticated XML-RPC scheme addresses the possible security attacks.

3.5.1 Replay Attack

One way to break a cryptography system is to simply resend a previously observed valid message. This attempt is known as a replay attack and to avoid it a receiver must be able to recognize a stale message, despite its previous validity. Of course, in our scheme, both parties must agree on a mechanism for determining the next expected request identifier, and note that since a new connection identifier is assigned for every login session, an adversary can perform this attack only when the player is still logged in.
A simple message count is sufficient for our purpose. This approach does not require a lot of memory to determine if a received message has been seen previously. However, the implementer of our protocol is free to choose any mechanism to generate a unique request identifier.

3.5.2 Message Spoofing

A spoofed message is an adversary generated message that has a valid message authentication code, causing a receiver to incorrectly accept the message as an authentic message from a legitimate sender. This is easily achieved if the XML payload is simply check-summed using MD5. Our protocol prevents this attack by introducing the *webToken* as a private key. The client’s *webToken* is never transmitted over the network. Therefore, an adversary who wants to spoof a message must somehow be able to infer the user *webToken*. However, the 128-bit long *webToken* is considered infeasible to brute force even for today’s available computing power.

3.5.3 Man-in-the-Middle Attack

Man-in-the-middle attack is an attack performed by an adversary who successfully plants itself as a communication intermediary between the client and the server. The adversary intercepts a message and then retransmits it to the receiver with or without modification. This attack is irrelevant in our scheme for two reasons. First, the adversary cannot modify the intercepted message unless it knows the secret *webToken*. Second, while it is true that the adversary can learn about the communication by intercepting messages, so can anybody since the messages are not encrypted.
3.5.4 Security of MD5

Standard MD5 is not considered secure anymore because it is vulnerable to collisions [30], i.e. the construction of different messages with the same MD5 hash. One technique to make the hash computation stronger is to employ a salt. A salt [5] is random bits that are used as one of the inputs to the hash generation algorithm. In the HTTP Digest Access Authentication, the nonce value acts as a salt during the authentication process. Consequently, no threat on the scheme by exploiting the MD5 collision techniques has been reported so far. This a good indication for our Authenticated XML-RPC scheme because our protocol follows a similar method of that used by the HTTP Digest Access Authentication. In our protocol, the webToken functions as a salt as it is unique for every user. However, if MD5 is still deemed insecure, it can be replaced by other stronger algorithms, such as SHA-2, without modifying the protocol.

3.5.5 Security of the Login Protocol

By examining the login protocol, one can see that the getPID web service is prone to denial-of-service (DoS) attack. In a DoS attack, some adversary swamps the server with a huge amount of requests which render the service unavailable. However, the getPID is a simple procedure thus it is easy to make it highly scalable. The server can also impose a limit on the number of getPID requests that can be submitted in a certain time period. For the connect procedure, the server verifies the login-session to avoid replay attack. A desirable characteristic of a login-session is that it should be easy to generate and verify. The encrypted server time is a good candidate for this purpose, since this approach does not require any memory. Verification at
the server is done simply by decrypting the login-session and comparing it with the current server time. The login session becomes invalid when the elapsed time exceeds a predefined time window. As discussed earlier, the client also issues a nonce to test the authenticity of the server that it is corresponding with.

3.5.6 Password Storage at the Server

User passwords should be stored in encoded form instead of in plain-text at the server for many security reasons. With this in mind, our scheme keeps the user password in the hashed form, that is \textit{webPassword}. Then, we use this hashed password to derive the \textit{webToken}. Both \textit{webPassword} and \textit{webToken} values can be stored by the server to amortize the computation cost. To improve security, we recommend that the \textit{webPasswords} and the \textit{webTokens} are physically stored separately. The user \textit{webTokens}, constantly used for authentication, should be stored in the production database or even in shared memory to speed up the authentication operation. The \textit{webPassword}, however, does not serve any purpose if the \textit{webToken} are already pre-computed and stored for all users. Hence, user \textit{webPasswords} should be kept offline.

The goal of this arrangement is to minimize user inconvenience when the system security is compromised or needs modification. For instance, intruders may gain access to the server and obtain the list of \textit{webPasswords}. In order to mitigate this, the game company only needs to fix the security hole and regenerate the \textit{webPasswords} using a new \textit{token_string}. It is also possible that the current scheme needs redesign. Similarly, the developer can derive any user password related information from the
webPassword. All these can be done without troubling the users to supply new passwords.

3.6 Enabling Information Privacy

An encryption scheme can be added into the Authenticated XML-RPC scheme. For secure message transmission, we suggest the encrypt-then-authenticate approach [20], that is encrypt the XML data and then compute the MAC. The login protocol may be extended as well to support encryption key negotiation if needed. Following the basic principals of cryptography, the key for encryption must be chosen judiciously, that is it must be independent from webToken, as the two keys serve different goals.

Alternatively, HTTPS is still an acceptable choice if the XML-RPC traffic is transmitted on a persistent connection. HTTPS already provides a message authentication feature, thus, the MAC tag can be stripped from the messages. However, the messages still can benefit from the <qReqInfo> elements as they are useful in keeping track of the communication between a client and the server.
CHAPTER 4
SPEX: Server-Push Enabled XML-RPC

4.1 Motivation

As we have discussed in Section 2.3, a lobby middleware provides three communication services: request-response service, server send service, and server multicast service. The request-response service works well with HTTP because it matches the request-response protocol. However, the HTTP request-response communication model poses a major challenge in supporting the server send or multicast service. These two server services are used by the server to propagate updates to clients in real-time at arbitrary time-points. The main question is how can we enable the server to push traffic using the request-response protocol?

A simple solution is to set the client to periodically poll the server, asking for updates. Such a technique causes an inevitable trade-off between application responsiveness and bandwidth consumption. The frequency at which a client polls the server adds an additional time delay to client updates. On average, updates are delayed by half the length of the polling interval and this delay must be kept as low as possible in order to achieve good real-time interactivity. On the other hand, increasing the polling frequency unnecessarily strains the server capacity and the network bandwidth. This polling technique will not give satisfactory performance for a web-based game lobby.
In the past, several techniques have been proposed to avoid the trade-off between responsiveness and bandwidth, such as Comet and XMPP BOSH. They offer low latency and yet consume less bandwidth than polling. Comet uses the HTTP chunked transfer encoding so that the client only needs to send one request in order to receive a data stream from the server. In contrast, XMPP BOSH adopts a long-polling mechanism. It uses synchronous HTTP request/response pairs where the server does not respond immediately to a request when there is no data, but waits until real data is available. Both techniques are good candidates for the lobby middleware.

We estimate that Comet is more efficient than BOSH. However, the chunked HTTP responses employed in Comet are not compatible with all HTTP proxy servers. It is known that some proxy servers buffer partial HTTP responses until the full response is available. This means that the responsiveness of a Comet data stream cannot be guaranteed. Some proxy servers also limit the duration of HTTP responses. BOSH overcomes these problems by using multiple request/response pairs. In addition, BOSH can support both HTTP/1.0 and HTTP/1.1 whereas Comet is restricted to HTTP/1.1 transport.

4.2 Design Overview

Thus, considering the merits of Comet and BOSH, we propose in this thesis a novel Server Push Enabled XML-RPC(SPEX) protocol which is based on the BOSH technique. Similar to BOSH, SPEX uses the long-polling technique. The SPEX client sends a special type of RPC request as a cue to the SPEX server of its willingness to wait for the SPEX server to push information. In SPEX, the special RPC request is denoted as a `getUpdates` RPC request. The SPEX server will reply to
the `getUpdates` RPC request only when it has data to push to the SPEX client. Unfortunately, allowing the SPEX server to hold the RPC request indefinitely makes both parties unable to detect if the other end is still active. Therefore, the SPEX server should respond to the client `getUpdates` RPC request after a certain period of time (called `requestTimeout` in this thesis), even if there is no data available. Upon receiving the `getUpdates` RPC response from the SPEX server, the SPEX client must immediately send again a new `getUpdates` RPC request to the SPEX server. This mechanism effectively functions as `keep-alive` or `ping` messages for both the SPEX server and the SPEX client. The SPEX’s server push mechanism is depicted in Figure 4–1.

![Figure 4–1: Server push mechanism in SPEX](image)

If the SPEX server does not receive a new `getUpdates` RPC request from the SPEX client within a specified time period (called `waitTimeout`) since returning the previous `getUpdates` RPC response, the SPEX server deems the communication

38
with the SPEX client broken. Similarly, the SPEX client considers that the communication has broken down if it does not receive a `getUpdate` RPC response from the SPEX server within a certain time period (called `responseTimeout`) since it last sent the `getUpdates` RPC request. The length of `responseTimeout` is determined by the length of `requestTimeout` and the network round-trip time (RTT). In this thesis, we do not provide any mechanism to recover from communication timeout. In such a situation, we assume that the client application can reinstate itself with the help of the server application when the communication is re-established (application level recovery).

Similar to BOSH, the SPEX server returns each RPC response using a complete HTTP response. The SPEX server can push an update to the SPEX client only when it has received a `getUpdates` RPC request to which it has not replied yet. Otherwise, the SPEX server will have to temporarily queue the client updates, as they become available, until a new `getUpdates` request arrives. This strategy prevents the proxies from holding the HTTP responses.

The server push traffic is preferably transmitted over an HTTP/1.1 connection which allows a series of HTTP requests/responses to be sent over a single persistent HTTP connection. This way, there is no delay incurred from TCP connection setup and tear down for every request/response pair.

SPEX must also support the standard XML-RPC functionality, that is executing RPC requests sent by the client application to the server (refer to Figure 2–7). Hereafter, an RPCs originating from the client application (lobby client) is referred to as `application RPC`. The SPEX client cannot use the connection used by the server
push traffic (getUpdates RPC) for transmitting the application RPC request to the SPEX server. Instead, the SPEX client can create a new HTTP connection (out-of-band channel) for each application RPC request it sends. This protocol is hereafter referred as SPEX Mono.

Alternatively, the SPEX client can setup another persistent HTTP connection with the SPEX server, over which it can send the application RPC requests. Having a second persistent connection again saves the overhead of setting up a new connection for each application RPC request.

Nevertheless, a player’s behavior in a game lobby does not guarantee that the lobby client generates RPC requests on a regular basis. For example, when a player is idling [8], there might be considerable time periods where the SPEX client does not send any application RPC request to the SPEX server. On these occasions, the idle time can possibly exceed the preset network connection timeout, and various network intermediaries, for example, firewalls and proxies, and the server may consider the connection as broken. We remedy this problem by sending the getUpdates requests on the two persistent connections alternately. As the getUpdates RPC have a keep-alive mechanism incorporated, both connections are maintained. When an application RPC request needs to be sent, the client simply uses the connection that is currently not in use by the getUpdates RPC mechanism. We refer to this solution as SPEX Duo.

There is a limit on the number of physical connections that a server can support. Consequently, a SPEX client should not have the latitude in opening an arbitrary number of connections as it wishes. In SPEX Duo, the limit is explicitly set to two
connections per client. We advocate that the same number should be imposed in
SPEX Mono as well. That is, a Mono client has one connection for server push
traffic and, at most, one out-of-band connection for sending an application RPC
request. For this to work efficiently, we have to assume that RPC execution time
by the server application is negligible so that the SPEX client does not need to wait
too long before it can send the next application RPC request. For application RPC
requests that consumes a lot of processing time, the server can simply return an
acknowledgment response and continue processing the RPC request. The result can
be eventually returned asynchronously by pushing it as an update to the client.

Comparing the Mono and the Duo, the Mono is a simple protocol where the
server push mechanism is an independent extension to the existing XML-RPC frame-
work. However, every application RPC suffers from the connection setup. In terms
of application RPC execution, the Duo should perform better than Mono at the ex-
 pense of having more complicated connection management. The Duo has to maintain
two persistent connections even if there is no application RPC traffic.

4.3 SPEX Framework

The SPEX framework is shown in Figure 4–2. SPEX is an extension of XML-
RPC; hence, it still uses the XML-RPC infrastructure. SPEX still adheres to the
XML-RPC specification, for example, it maintains the request-reply protocol and
uses the XML-RPC standard for serializing RPC requests and responses.

At the client side, the SPEX client provides two services: execute-rpc and
update-callback. The execute-rpc service is used by the client application to
make RPC requests to the server by specifying the method name and parameters.
The `execute-rpc` synchronously returns the RPC execution result to the client application. The SPEX client handles the server push traffic in a way that is transparent to the client application. It internally injects the `getUpdates` RPC request as part of the server push mechanism. When an update is received from the SPEX server, the SPEX client uses a callback mechanism, called `update-callback`, to pass the updates to the client application. Since the `getUpdates` RPC is reserved for the server push mechanism, the client application never makes a `getUpdates` RPC request.

The SPEX server offers three services to the server application: `dispatch`, `send`, and `mcast`. The dispatch service is a service that is inherited from the standard XML-RPC. When the SPEX server receives an application RPC request, it forwards the method name and parameters to the server application to be executed. The result of the execution is returned to the SPEX client as an RPC response. Similar to the SPEX client, the SPEX server hides the server push mechanism from the
server application. It handles the processing of \texttt{getUpdates} RPC internally without involving the server application. The SPEX server provides two communication services for the server to disseminate updates to clients. The \texttt{send} service is a unicast communication that sends an update to a specified client. In the situation when the server application needs to propagate an update to multiple clients at once, the server application can use the \texttt{mcast} service and provide the list of clients.

The five services offered by SPEX framework realize all three communication services that a lobby middleware should provide (as discussed in Section 2.3). The \texttt{execute-rpc} and \texttt{dispatch} implement the \textit{request-response} communication service. The \texttt{send} and \texttt{update-callback} realize the \textit{send} communication service. Finally, the \texttt{mcast} and \texttt{update-callback} realize the \textit{multicast} communication service.

In the following we describe both SPEX Mono and SPEX Duo in more details by using the SPEX framework as reference. We will not discuss the \texttt{mcast} service as we assume that the \texttt{mcast} service is implemented based on the \texttt{send} service by transmitting the update to each SPEX client individually (refer to our discussion in Section 2.3).

4.4 SPEX Mono Algorithm

4.4.1 Connection Setup

The SPEX Mono protocol relies on having one dedicated persistent HTTP connection between the client and the server, but there is no notion of connection or session in the XML-RPC specification. By default, an RPC request/response pair uses an HTTP/1.0 connection, thus, the specification does not outline the signaling mechanisms for establishing, maintaining, or terminating a connection. However,
persistent connection management is critical in the SPEX protocol. For this purpose, we adapt the two-phase login protocol of the Authenticated XML-RPC scheme (presented in Section 3.4) to include setting up the persistent connection for our SPEX Mono (see Figure 4–3 for illustration). In the following, we discuss how we adapt each of the login procedures of Authenticated XML-RPC:

1. getPID. This procedure remains unchanged. The client executes this procedure to request the user’s pid and a login-session from the server.

2. connect. The client then establishes an HTTP/1.1 connection with the server to send the connect RPC request. The connect procedure registers this connection with the server. The connection is retained by the server as the persistent connection for pushing data to the client. For that purpose, the server maintains a connection dictionary (ConnectionDict) data structure to keep all client persistent connections indexed by their pids.

![Figure 4–3: SPEX Mono login protocol](image)

Logically, the server should not process any RPC request from clients that are not logged into the game yet, which can be verified using the pid and cid information.
Therefore, in addition to using the login protocol of Authenticated XML-RPC, every SPEX message also includes the `<qReqInfo>` structure. Later on, we will show how SPEX can also benefit from the other request information elements beside the pid and the cid.

### 4.4.2 Processing Application RPC Requests and Server Updates

The SPEX Mono client is summarized in Figure 4–4. The `send-request` method at lines 1 and 9 in the pseudocode is used to marshall and send an RPC request. The first and second parameters of `send-request` are the remote method name and the method parameters; while the third parameter contains the connection to be used for this communication. The `get-response` method at lines 4 and 16 is a blocking method that waits until an RPC response is received at the specified connection. When an RPC response is received, the `get-response` method returns the unmarshalled RPC response.

Once the login has successfully completed, the SPEX client directly sends a `getUpdates` RPC request to the server to start the server push mechanism. The SPEX client also has to ensure that the SPEX server has a buffered `getUpdates` RPC request most of the time throughout the lobby session. To achieve this, the client can start a background loop to handle the update traffic from the server. Once it receives a `getUpdates` RPC response from the server, it passes the updates to the client application, if any, using the `update-callback` and sends immediately a new `getUpdates` RPC request. However, if there is no `getUpdates` RPC response received from the server within the `responseTimeout` duration, the SPEX client considers the
Code for SPEX Mono client:

background loop for retrieving updates from the server:

**Input:**  \( pconn \) ← the persistent HTTP/1.1 connection setup during login

1. **send-request**(“getUpdates”, ⊘, \( pconn \)) // initialize the server push mechanism
2. set responseTimeout timer
3. **while true do**
4. \( updates \) ← **get-response**(\( pconn \)) // wait for updates from server
5. unset responseTimeout timer
6. **if** \( updates \neq \emptyset \) **then**
7. pass \( updates \) to the client application using update-callback
8. **end if**
9. **send-request**(“getUpdates”, ⊘, \( pconn \)) // send new getUpdates request
10. set responseTimeout timer
11. **end while**

**upon expiry of responseTimeout timer:**

12. notify the client application about the communication break down
13. terminate

**upon receiving execute-rpc from client application:**

**Input:**  \( methodName \) ← the methodName of the application RPC

**Input:**  \( params \) ← the parameters of the application RPC

14. \( conn \) ← open a new HTTP/1.0 connection
15. **send-request**(\( methodName \), \( params \), \( conn \))
16. result ← **get-response**(\( conn \))
17. **close**(\( conn \)) // close the connection
18. **return** result to client application

Figure 4–4: SPEX Mono client protocol

communication to have broken down. When this happens, the SPEX client notifies the client application and terminates.

As discussed earlier, the SPEX Mono client transmits the application RPC request to the server using an out-of-bound channel. Thus, when the client application invokes the **execute-rpc** service, the SPEX client first has to open a new connection with the SPEX server before sending the application RPC request (line 14).
The connection is torn down (line 17) after the SPEX client has received the RPC response.

The pseudocode for the SPEX Mono server is outlined in Figure 4–5. The send-response method at line 10, 14, 23 marshalls the update, given as the first parameter, into the RPC response and sends it using the connection specified in the second parameter. The first part of the pseudocode handles the getUpdates RPC request. When the SPEX server receives a getUpdates RPC request from a SPEX client, it first checks if there is any queued update for that client. The server stores all queued updates in a list called QueuedUpdates. If there is no queued update found for the SPEX client, the SPEX server buffers the getUpdates RPC request in a list called BufferedRequest. Otherwise, the server would send the queued updates to the SPEX client by directly responding to the getUpdates RPC request. After sending the getUpdates RPC response, the SPEX server resets the waitTimeout timer for that SPEX client and waits for that SPEX client’s next getUpdates RPC request.

The application RPC requests received from out-of-band channels are essentially the same as the RPC requests in the standard XML-RPC, thus there is no difference in handling them with the existing XML-RPC protocol. The pseudocode for handling the application RPC request is shown in the second part of Figure 4–5. The application RPC request is dispatched to the server application to be executed. The server application returns the result of the execution to the SPEX server to be propagated to the SPEX client as an RPC response.

The last part of the pseudocode describes how the Mono server handles the send service. To send an update to a SPEX client, the SPEX server first checks
Code for SPEX Mono server:

Global variables:
- **BufferedRequest**: a list to hold clients’ buffered getUpdates RPC request
- **QueuedUpdates**: a list to hold clients’ queued updates
- **ConnectionDict**: a dictionary to hold all client connections indexed by the pids

upon receiving a getUpdates RPC request from a SPEX client:

**Input**: `updtRequest ←` the getUpdates request

**Input**: `spid ←` the sender’s pid

1: unset the `waitTimeout` timer of this client
2: `updates ←` retrieve queued updates for this `spid` from `QueuedUpdates`
3: if `updates = ⊘` then
4: // no queued updates for this client, so buffer this request
5: set the `requestTimeout` timer of this client
6: insert `(spid, updtRequest)` tuple to `BufferedRequest`
7: else
8: // there are queued updates for this client
9: `pconn ←` retrieve the persistent connection for this `spid` from `ConnectionDict`
10: `send-response(updates, pconn) // return the updates using pconn`
11: set the `waitTimeout` of this client
12: end if

upon receiving an application RPC request from a SPEX client:

**Input**: `rpcRequest ←` the unmarshalled application RPC request

**Input**: `conn ←` the connection over which the RPC request is received

13: `result ← dispatch(rpcRequest) // dispatch the method execution to server application`
14: `send-response(result, conn) // return the execution result`

upon receiving `send` from the server application:

**Input**: `rpid ←` the pid of the client to receive the update

**Input**: `update ←` the update for the client

15: `updtRequest ←` remove buffered getUpdates RPC request for this `rpid` from `BufferedRequest`
16: if `updtRequest = ⊘` then
17: // no buffered getUpdates RPC request for this client
18: insert `(rpid, update)` tuple to `QueuedUpdates` // enqueue the update
19: else
20: // use the buffered getUpdates RPC request to send the update to the client
21: unset the `requestTimeout` timer of this client
22: `pconn ←` retrieve the persistent connection for this `rpid` from `ConnectionDict`
23: `send-response(update, pconn) // send the update using pconn`
24: set `waitTimeout` timer of this client // wait for next request
25: end if

Figure 4-5: SPEX Mono server protocol
if the SPEX client has a buffered `getUpdates` RPC request in the `BufferedRequest` list. If there is no buffered `getUpdates` RPC request, the update is enqueued into the `QueuedUpdates` list. Otherwise, the SPEX server can transmit the updates by responding to the buffered `getUpdates` RPC request.

Figure 4–6 shows how the Mono server handles the expiry of `requestTimeout` and `waitTimeout`. The expiry of the `requestTimeout` of a client means the buffered `getUpdates` RPC request must be answered, even though there is no update to be transmitted. Thus, the Mono server responds to the `getUpdates` RPC request with an empty RPC response.

When the Mono server detects a `waitTimeout` expiry of a particular client, meaning that the server has not received a timely new `getUpdates` RPC request from a client, the SPEX server regards the communication with the SPEX client to have terminated unexpectedly. In this situation, the SPEX server notifies the server application that the player is no longer reachable. The SPEX server also needs to remove the SPEX client’s persistent connection from the `ConnectionDict` and its queued updates in the `QueuedUpdates` list.

4.5 SPEX Duo Algorithm

4.5.1 Connection Setup

As discussed earlier, a SPEX Duo client maintains two persistent HTTP connections with the server. To establish the two persistent connections, we further extend the connection setup protocol used in Mono by adding a third step: the client creates and registers a second persistent connection by calling the `addConnection` RPC. Figure 4–7 illustrates the login mechanism for SPEX Duo. In the figure, messages that
**Code for SPEX Mono server:**

**upon expiry of a client’s requestTimeout timer:**

**Input:** \(\text{pid} \leftarrow \text{the pid of the getUpdates RPC request sender} \)

1: \(\text{pconn} \leftarrow \text{retrieve the persistent connection for this pid from ConnectionDict} \)

2: remove the buffered getUpdates RPC request for this \(\text{rpid from BufferedRequest} \)

3: \texttt{send-response}(⊘, \texttt{pconn}) // return empty response to the SPEX client

4: set the \texttt{waitTimeout} timer of this client

**upon expiry of a client’s waitTimeout timer:**

**Input:** \(\text{pid} \leftarrow \text{the pid of the SPEX client} \)

5: \(\text{pconn} \leftarrow \text{retrieve the persistent connection for this pid from ConnectionDict} \)

6: remove \((\text{pid}, \text{pconn})\) from \texttt{ConnectionDict}

7: remove all queued updates for this \text{pid} from \texttt{QueuedUpdates}

8: notify the server application about the communication break down for this \text{pid}

9: \texttt{close(\text{pconn})} // close the persistent connection

---

Figure 4–6: SPEX Mono server protocol for handling expiry of requestTimeout and waitTimeout

are sent using the first and the second persistent connection are prefixed with \(C1\) and \(C2\) respectively. This convention will also be used for the rest of the discussion.

### 4.5.2 Processing Application RPC Requests and Server Updates

A SPEX Duo client needs to keep both persistent connections alive even if there is nothing to be sent to or from the server. The protocol is designed for the SPEX client to toggle between the two connections in sending the getUpdates RPC requests so as to maintain traffic on both connections (Figure 4–8). Hence, unlike Mono, none of the connections is dedicated for the server push traffic. Instead, at any one time, one of the connections serves this purpose and the other is for the SPEX client to send application RPC requests to the server.

To further reduce the getUpdates RPC traffic, we allow the server to swap the connection used by the buffered getUpdates RPC request with the connection used
by a newly received application RPC request. This means, before the SPEX server returns the application RPC response, it first checks if there is a buffered `getUpdates` RPC request for that client. If there is, it sends the application RPC response through the connection from which it received the `getUpdates` RPC request. It also resets the `requestTimeout` for the buffered `getUpdates` RPC request and adjusts the connection through which the RPC response for this buffered `getUpdates` RPC request should be sent. If there is no buffered `getUpdates` RPC request, the server simply returns the application RPC response using the original connection from which it received the application RPC request. To support this connection swapping, the client must be able to reconcile every RPC response with its corresponding RPC request. The client can rely on the request identifier (`rid`) information included in the RPC request and the RPC response (refer to Section 3.3) to do this matching. Figure 4–9 shows an example of this mechanism where the connections get swapped.
Figure 4–8: SPEX Duo client alternately sends `getUpdates` RPC requests between the two persistent connections while handling an application RPC request. In this example, the server holds the client’s `getUpdate` RPC request at the first persistent connection. Before the `request-Timeout` expires, the client sends an `Account.PurchaseCredit` RPC request to the server using the second persistent connection. Once the `Account.PurchaseCredit` RPC request is executed, the server can return the result on the first connection instead of using the second connection. After that, the server pushes the next update to the client on the second connection. Note that, because of this connection swapping mechanism, if there are enough application RPC requests from the client, the `getUpdates` RPC requests never have to be answered with an empty RPC response.

The pseudocode for the SPEX Duo client for handling the server push traffic is summarized in Figure 4–10. The `send-request` method at line 2 and 18 is responsible for sending an RPC request using the specified request identifier and persistent connection. Thus, a new request identifier has to be generated before sending an RPC request. After the Duo client has completed the `addConnection` RPC, it sends
Figure 4–9: SPEX Duo server returning an RPC request by swapping the connections the first `getUpdates` RPC request on the first persistent connection to initialize the server push mechanism.

In contrast to the Mono Client, the background loop for Duo listens on both persistent connections (line 5). Thus, the background loop handles RPC responses of `getUpdates` RPC request and application RPC request from `execute-rpc`. Since the SPEX server may swap connections when returning an application RPC, the Duo client has to cross check the `rid` in the received RPC response with the RPC request that has not been answered yet (called `pending RPC`), if any. In the pseudocode, the `rid` of the pending application RPC request is stored in the `PendingRPC` global variable. If the `rid` of the RPC response matches `PendingRPC` (line 6), then it is a response to the pending application RPC from `execute-rpc`, otherwise, it is actually the response to the `getUpdates` RPC request. The background loop passes the result of the application RPC to the `execute-rpc` process by putting the RPC execution result in the `RPCResult` global variable. The condition variable `RespCond` is used to signal to the `execute-rpc` process that the result is available. If the RPC
Code for SPEX Duo client:

Global variables:
- PendingRPC: the rid of the pending RPC request (if any)
- RPCResult: returned response by the server for the pending RPC request
- RespCond: a condition variable to signal that the RPC response is available

Background loop for retrieving updates from the server:
Input: pconn1 ← the first persistent connection
Input: pconn2 ← the second persistent connection
1: rid ← new unique request identifier
2: send-request("getUpdates", ⊘, rid, pconn1) // initialize the server push traffic
3: set responseTimeout timer
4: while true do
5: response ← get-response([pconn1, pconn2]) // wait for an RPC response from the server
6: if response.rid = PendingRPC then
7: // it’s a response to the pending RPC Request
8: RPCResult ← response.result // pass the result to execute-rpc execution
9: signal RespCond // signal that the response is available
10: else
11: // it’s a getUpdates response
12: unset responseTimeout timer
13: if updates ≠ ⊘ then
14: pass response.result to the client application using update-callback
15: end if
16: rid ← new unique request identifier
17: pconn ← determine next outgoing connection
18: send-request("getUpdates", ⊘, rid, pconn) // send new getUpdates request
19: set responseTimeout timer
20: end if
21: end while

Upon expiry of responseTimeout timer:
22: notify the client application about the communication break down
23: terminate

Figure 4–10: SPEX Duo client protocol for the server push mechanism
response is actually the response to the `getUpdates` RPC request, then the response carries the server updates that should be forwarded to the client application using the `update-callback`.

The Duo client handles the `responseTimeout` in the same fashion as Mono does. The Duo client regards the communication with the server as broken if there is no response received from a `getUpdates` RPC request within the `responseTimeout` period. Consequently, the SPEX client notifies the client application about the failure and initiates termination.

Figure 4–11 shows how the Duo client handles an `execute-rpc` invocation from client application. Before the application RPC request is sent to the SPEX server, the process assigns the `rid` value of the RPC request to the `PendingRPC` global variable. This is necessary so that the background loop can return the result of the application RPC back to this process. After sending the RPC request to the server, the process waits for the signal from the background loop for the execution result by waiting on the `RespCond` condition variable. When the process receives the signal, it retrieves the execution result from `RPCResult` and clears `PendingRPC`.

Determining which connection should be used for sending an RPC request (line 17 of Figure 4–10 and line 2 of Figure 4–11) is a very critical step. At any time, the SPEX server only listens on one of the two client connections because of the toggling mechanism. Consequently, the client and the server must always be in agreement on which connection the next request must be sent. This decision depends on which connection the server has sent the last response and from which connection the client received the last response on. The decision becomes tricky when the server...
Code for SPEX Duo client:

upon receiving execute-rpc from client application:

Input: methodName ← the methodName of the application RPC
Input: params ← the parameters of the application RPC

1: rid ← new unique request identifier
2: pconn ← determine next outgoing connection
3: PendingRPC ← rid // set PendingRPC with the rid
4: send-request(methodName, params, rid, pconn) // send the request using pconn connection
5: wait RCond // wait for server response
6: PendingRPC ← ⊘ // clear PendingRPC
7: return RPCResult

Figure 4–11: SPEX Duo client protocol for processing an application RPC

returns two RPC responses through both connections at almost the same time. In this case, the client may actually receive the responses in the reverse order from how they are being sent. This is possible because both responses are transmitted over two separate connections, and thus may be routed separately. This situation may result in server waitTimeouts and client responseTimeouts because the client sends the next request from one connection and the server is listening at the other connection.

In order to keep our code exposition simple, we do not include the mechanism to overcome the above situation in the Duo client pseudocode (Figure 4–10 and Figure 4–11). Instead, the following discussion elaborates how this situation can be resolved. To prevent the above scenario, the SPEX server maintains a response counter (rc) for every SPEX client and includes this count information in the RPC responses sent to the SPEX client (refer to Section 3.3). The SPEX client can then infer the actual sequence of the RPC responses based on the server response count.
information. Figure 4–12 depicts an example of this scenario (similar to the example in Figure 4–9), showing how the SPEX client recovers from it. In this example, \( rc \) is set to 0 and the SPEX server holds a \texttt{getUpdates} RPC request from the SPEX client at the first connection. As the game plays, the SPEX server receives a \texttt{Account.PurchaseCredit} RPC request from the SPEX client. Not long after the SPEX server returns the \texttt{Account.PurchaseCredit} RPC response on connection one (\( rc = 1 \)), an update for the client is received and, thus, pushed to the client on connection two (\( rc = 2 \)). Due to some network delay, the \texttt{getUpdates} RPC response reaches the SPEX client first. Since it shows a response count of 2, the client knows that there is another incoming message on the first connection and it should send the next \texttt{getUpdates} on the first connection instead of on the second connection.

Figure 4–12: SPEX Duo handling the special case in simultaneous RPC responses

Figure 4–13 outlines the pseudocode for the SPEX Duo server. The Duo server allocates one process per client. The process is in a loop waiting for an incoming RPC request from the client at one of the two connections. The server uses the RPC request method name to determine the type of the RPC request, that is \texttt{getUpdates}
RPC request or application RPC request. The server has the `processGetUpdates` and `processRPC` helper methods (shown in Figure 4–14) to process the `getUpdates` RPC request and the application RPC request respectively. After the received RPC has been processed, the client process starts listening at the next persistent connection.

The `processGetUpdates` processes `getUpdates` RPC in the same way as the Mono server does. The `getUpdates` RPC request is answered immediately if there are queued updates for the client. Otherwise, the server buffers the `getUpdates` RPC request.

The mechanism to swap connections is shown in the `processRPC` pseudocode. The `processRPC` method first dispatches the application RPC to the server application for execution. To return the execution result, it checks the availability of the buffered `getUpdates` RPC request in the `BufferedRequest` list. The connection swapping is only performed when there is a buffered `getUpdates` RPC request.

The Duo server handles the `send` service in a similar way to Mono (refer to the bottom part of the pseudocode in Figure 4–13). The update is sent to the client by responding to the buffered `getUpdates` RPC request. If there is no buffered `getUpdates` RPC request, the update is queued in the `QueuedUpdates` list.

Figure 4–14 shows how the SPEX Duo server handles the expiry of `requestTimeout` and `waitTimeout`. The Duo server also handles both timeouts in a similar manner as the Mono server. When the `requestTimeout` of a client expires, the SPEX server responds to the buffered `getUpdates` RPC request with an empty RPC response and starts the `waitTimeout` timer.
Code for SPEX Duo server:

Global variables:
- `BufferedRequest`: a list to hold clients' buffered getUpdates RPC request
- `QueuedUpdates`: a list to hold clients' queued updates
- `rc`: counter to track response count

background loop for processing a client’s RPC requests:
1. `pconn` ← the first persistent connection of the client
2. set the `waitTimeout` timer for this client
3. while true do
   4. `request` ← `get-request`(pconn) // wait for client RPC request
   5. if `request.type` = “getUpdates” then
      6. processGetUpdates(`request`, `pconn`)
   7. else
      8. processRPC(`request`, `pconn`)
   9. end if
   10. `pconn` ← the next persistent connection // listen at the next connection
   11. end while

upon receiving send from the server application:
Input: `rpid` ← the pid of the client to receive the update
Input: `update` ← the update for the client
11. `updtRequest` ← remove the buffered getUpdates RPC request of this `rpid` from `BufferedRequest`
12. if `updtRequest` = ⊘ then
13. // no buffered getUpdates RPC request for this client
14. insert (`rpid`, `update`) tuple to `QueuedUpdates`
15. // queue the update
16. else
17. // use the buffered getUpdates RPC request to send the update to the client
18. `rc` ← `rc` + 1
19. send-reply(`update`, `request.rid`, `rc`, `request.conn`)
20. set the `waitTimeout` timer of this client
21. end if

Figure 4–13: SPEX Duo server protocol
Code for SPEX Duo server:

\[\text{processGetUpdates}(\text{updtRequest}, \text{pconn})\]
1: spid ← the sender’s pid
2: updates ← retrieve queued updates for this spid from QueuedUpdates
3: if updates = \(\emptyset\) then
4: // no queued updates for this client
5: set the requestTimeout timer of this client
6: updtRequest.conn ← pconn // remember the outgoing connection
7: insert (spid, updtRequest) to BufferedRequest // buffer this request
8: else
9: // there are queued updates for this client
10: rc ← rc + 1
11: rid ← get the rid of updtRequest
12: send(update, rid, rc, pconn)
13: set waitTimeout timer of this client
14: end if

\[\text{processRPC}(\text{rpcRequest}, \text{pconn})\]
15: result ← dispatch(rpcRequest)
16: rid ← get the rid of rpcRequest
17: rc ← rc + 1
18: updtRequest ← remove buffered getUpdates request for this pid in BufferedRequest
19: if updtRequest = \(\emptyset\) then
20: // there is no buffered request
21: send-reply(result, rpcRequest.rid, rc, pconn)
22: else
23: // there is a buffered getUpdates request, so swap the connections
24: send(result, rpcRequest.rid, rc, brequest.conn)
25: updtRequest.conn ← pconn // update the outgoing connection for this request
26: reset requestTimeout timer of this client
27: insert (spid, updtRequest, btimeout) tuple to BR
28: end if

Figure 4–14: SPEX Duo server protocol helper methods
When the SPEX server detects a waitTimeout expiry for a particular SPEX client, the SPEX server assumes that the communication with the SPEX client to have terminated unexpectedly. In such a situation, the SPEX server notifies the server application about the termination and deletes any queued updates for this client.

**Code for SPEX Duo server:**

upon expiry of a client’s requestTimeout timer:

Input: pid ← the pid of the getUpdates RPC request sender

1: updtRequest ← remove the buffered getUpdates RPC request for this pid from BufferedRequest
2: send-reply(⊘, updtRequest.rid, rc, updtRequest.conn) // return empty response to the SPEX client
3: set the waitTimeout timer of this client

upon expiry of a client’s waitTimeout timer:

Input: pid ← the pid of the client

4: pconn1 ← the first persistent connection of the client
5: pconn2 ← the second persistent connection of the client
6: remove all queued updates for this pid from QueuedUpdates
7: notify the server application about the communication break down for this pid
8: close(pconn1) // close the persistent connections
9: close(pconn2)

Figure 4–15: SPEX Duo server protocol for handling expiry of requestTimeout and waitTimeout

### 4.6 Comparing SPEX Mono and Duo

In both Mono and Duo, the server push traffic is delivered using a persistent connection. In this respect, both protocols should have similar performance. The primary difference of the two protocols lies on how application RPC requests are delivered to the server. We can clearly see that Duo does not suffer from communication overhead as much as Mono, which has to open a new connection for every
application RPC request/response pair. It is feasible though for Mono to send the application RPC requests over a single persistent connection. But the Mono client ought to guarantee that an application RPC request is sent to the server periodically using this connection so that the connection is not considered inactive and, in turn, terminated by the server or any other network intermediaries that lie between the client and the server. In this case, the Mono protocol becomes less efficient than the Duo protocol because it has to send a dummy RPC request to the server periodically to simulate traffic when there is no actual application RPC request to be sent. Nevertheless, the efficiency of Duo compared to Mono also depends on the characteristics of the traffic that is being carried. For example, if most traffic is just the server pushing updates to the client and the client only occasionally or even rarely sends application RPC requests, then the efficiency gain of Duo over Mono diminishes as well.

Since the Duo has two persistent connections to be managed, Duo’s requestTimeout must be set at a shorter period than the Mono’s requestTimeout. The Mono’s requestTimeout reflects the preset network communication timeout, that is the maximum amount of time the connection can remain silent without being terminated by the network intermediaries. On the other hand, Duo has to ensure that there are communication over both persistent connections for the same timeout duration. So in the worst case, the time elapsed since the Duo client receives a getUpdates RPC response over a persistent connection until it sends a new getUpdates RPC request over the same connection should be within the network timeout period. Thus, the requestTimeout for Duo can be formulated as follows:
requestTime = network timeout - network RTT

Figure 4–16 illustrates further the relationship between the network timeout and the responseTimeout of Mono and Duo. Since Duo has shorter requestTimeout than Mono, the server push mechanism in Duo is less efficient than Mono in terms of the number of messages. However, considering the network RTT (around hundred milliseconds) is much smaller than the network timeout (tens of seconds or even several minutes), the performance difference is not significant.

In addition, the Duo also has the connection swapping mechanism which reduces the getUpdates RPC traffic. If the client application sends enough application RPC requests to the Duo server, the requestTimeout of the buffered getUpdates RPC request never expires, thus, the Duo server does not need to answer the buffered getUpdates RPC request with an empty message when there is no real updates to be pushed to the client. On the other hand, the Mono does not have such mechanism.
The \textit{requestTimeout} of the buffered \texttt{getUpdates} RPC request in Mono is never reset and must be answered with an empty RPC response when it expires.

\subsection{Comparing SPEX, Standard XML-RPC, and XMPP BOSH}

SPEX is an extension of the standard XML-RPC. SPEX still adheres to the specification outlined in the XML-RPC specification and is fully compliant with HTTP/1.0 or HTTP/1.1 as transport. The new feature that SPEX introduces to the XML-RPC framework is the server capability to push data to a client at the server discretion rather than waiting for the client to pull the data. To enable this, we have added the connection management feature, the long-polling mechanism, and the additional message trailer to the standard XML-RPC messages (the \texttt{<qReqInfo>} structure). As opposed to the standard XML-RPC, SPEX has a connection setup mechanism for establishing persistent connections between the client and the server. To keep track of the status of each connection, the SPEX server also needs to identify the sender of every request it receives, by using the information included in the new message trailer. Furthermore, SPEX reserves the \texttt{getUpdates} RPC to distinguish requests that can be buffered by the server from other RPC requests.

It is feasible for SPEX to be fully compliant with HTTP/1.0, although we have not discussed this in detail. The HTTP/1.0 compliant SPEX is very similar to Mono, only that it does not use a persistent connection for the server push mechanism. A fresh HTTP/1.0 connection has to be opened for every \texttt{getUpdates} RPC request/response pair which naturally hurts the responsiveness of the update traffic. Therefore, we do not recommend this approach.
Even though the design of SPEX is mainly inspired by BOSH, there are several significant differences between the two protocols. SPEX is inherently an RPC framework using the request-response protocol whereby every RPC request generates an RPC response that returns the result of the RPC execution. BOSH has a rather different communication model. An HTTP request received from a BOSH client carries a block of information to be pushed to the XMPP server and the HTTP request can be buffered to transmit updates to the client later on. BOSH also allows the client to send empty HTTP request to be buffered by the server. As a result, BOSH does not need a mechanism to indicate that a request can be temporarily held by the server.

The original BOSH design constrains the client not to open more than two HTTP connections to the server at the same time. In exchange, the server always returns an empty HTTP response to the HTTP request it has been holding on as soon as it receives a new HTTP request from the client even if there is no data for the client. The new HTTP request becomes the new buffered HTTP request for transmitting updates to the client on a later time. Putting the same mechanism into SPEX perspective means the application RPC request also functions as a \texttt{getUpdates} RPC request. Therefore, this mechanism is not viable in our SPEX framework. Instead, SPEX assumes that application RPC execution is negligible so that the SPEX client can send the next application RPC request quickly.

In addition, SPEX Duo also uses the connection swapping strategy (see Figure 4–9) which produces almost the same effect as the mechanism used by BOSH. The connection swapping strategy does not cause an unnecessary empty response and even extends the \texttt{responseTimeout} of the existing buffered \texttt{getUpdates} request.
In BOSH, most of the time data can be pushed immediately by the client or the server. However, if one of the end-points has just pushed some data then it will usually have to wait for a network round trip until it is able to push again. This limitation also applies to SPEX; particularly (a) after the SPEX server responds to a `getUpdates` RPC request and (b) after the SPEX client sends an application RPC request to the server. This problem can be remedied by enabling HTTP pipelining. If it is available, the SPEX client can make multiple concurrent RPC requests over a single HTTP connection without having to wait for the response of the previous RPC request. Therefore, nothing stops the SPEX client from sending application RPC request to the SPEX server immediately. The client can also ensure that the server is always buffering enough `getUpdates` requests such that even during burst of activity the server will never have to wait before pushing data.
So far we have introduced the design of SPEX and Authenticated XML-RPC. In this chapter, we describe an implementation of both protocols by adapting available XML-RPC implementations and the challenges encountered in this implementation. We use Python 2.5 [14] to implement the SPEX server and Java 6 [21] for the SPEX client. Our choice of the programming platforms is mainly driven by the eventual goal of our project, that is to integrate Mammoth [4], a Java based MMOG platform from McGill, with the Quazal Rendez-Vous game lobby which can be extended using Python. This setup also demonstrates the interoperability of XML-RPC in bridging heterogeneous systems. The portability of Python and Java applications is another factor which influenced our choice. In our implementation, we integrate the Authenticated XML-RPC scheme directly into our SPEX implementation.

The SPEX framework described in Section 4.3 defines the high-level architecture for our implementation. In the following, we describe the two SPEX implementations and then the implementation of the Authenticated XML-RPC scheme.

5.1 SPEX Server

Our SPEX server is built on top of a generic XML-RPC library implementation provided by Python 2.5. For our implementation, we are particularly interested in the SimpleXMLRPCServer module. We use this module as the basis for our SPEX server implementation. The SimpleXMLRPCServer is essentially a TCP server. The
main server process is always in a loop accepting incoming client connections on a designated TCP port. Once a connection with the client is established, the server instantiates a `SimpleXMLRPCRequestHandler` to process the incoming message, which is an HTTP POST request with XML-RPC payload. The server can be configured to process the request using (1) the main server process, (2) a newly spawned process, or (3) a newly created thread. By default, the XML-RPC request handler assumes the HTTP/1.0 mode, thus the connection is automatically closed once a response has been sent. When the mode is switched to HTTP/1.1, the request handler automatically waits for more incoming requests instead of terminating the connection. The request handler uses the `SimpleXMLRPCDispatcher` to unmarshal and dispatch the RPC request and also marshal the execution result. In the following two subsections, we will discuss how we extend the existing Python XML-RPC library to implement SPEX Mono and Duo server.

### 5.1.1 SPEX Mono

In SPEX Mono, clients establish a single persistent connection to receive updates from the server and submit application RPC requests using out-of-band channels, which are standard RPC requests in XML-RPC. The class diagram in Figure 5–1 gives the overview of our SPEX Mono server implementation.

A `SpexMonoServer` class is defined to encapsulate the implementation of the server, following the *façade* design pattern [16]. The `send` method implements the `send` service as defined in the SPEX framework. To invoke `send`, the server application only needs to provide the pid of the receiver and the update to be sent. We implement the `mcast` service by using the `send` service for every pid specified in the receiver...
list, and provide two implementations for the `mcast` service. The `iterativeMcast` method invokes the `send` method iteratively, whereas the `concurrentMcast` method does it concurrently. We provide the two different implementations because we want to see if performing the I/O operations concurrently can speed up the multicast.

The `SpexMonoServer` itself functions as the dispatcher. The server application registers the methods that are exposed for the RPC using the `register-function` method. The `dispatch` method performs the dispatch service by invoking the appropriate registered method as specified in the application RPC request.
The persistent connections for server push are abstracted by the `Connection` class. A `Connection` instance also holds the buffered `getUpdates` RPC request (`bufferedRequest`), if any, and a list to queue the updates (`updateList`). All `Connection` instances are stored in a global dictionary data structure, called `ConnectionDict`, using the client pids as the key for easy retrieval. A `Connection` instance is instantiated and stored in the `ConnectionDict` upon successful `connect` RPC execution.

The `Connection` class provides a `sendUpdate` method that actually implements the sending of an update to the client. Thus, when the `send` method of `SpexMonoServer` is called, it retrieves the `Connection` instance of the client from the `ConnectionDict` and invokes the `sendUpdate` method.

We employ two synchronization locks to preserve data consistency in a `Connection` instance: `requestLock` and `syncLock`. The `requestLock` is for synchronizing operations on `bufferedRequest` and `updateList`, whereas the `syncLock` is used to protect the rest of the `Connection` attributes. When the `sendUpdates` method is invoked it first acquires the `requestLock` to check the availability of the buffered `getUpdates` RPC request. If there is none, then the update is queued and the `requestLock` is released. Otherwise, the `syncLock` is acquired before releasing the `requestLock`, and only then the update can be sent to the client. This concurrency problem can actually be handled in a simpler way by employing only one lock to synchronize all operations. We use two locks to improve parallelism, allowing a thread to queue a client update while another thread is sending another update to the client.
As opposed to standard XML-RPC messages, SPEX messages carry additional request related information which is encoded using the <qReqInfo> XML structure in the messages. We adapt the SimpleXMLRPCRequestHandler by creating a subclass of it, called SPEXRequestHandler, to add the code for parsing the <qReqInfo> structure of the incoming requests and appending the appropriate <qReqInfo> to the outgoing responses. The send method of Connection also needs to append the <qReqInfo> in the response messages that is sent using this method.

In our implementation, we design the Mono server to create a new thread to process each application RPC request. Except for the connect RPC request, the child thread can close the connection and terminate after responding to the request. The connect RPC is different because this RPC request also registers the persistent connection with the server for receiving updates from the server. Accordingly, the connection should not be closed. To reduce thread creation and termination overhead; we also provide an implementation of a thread-pool in our module. The SpexMonoServer class can be easily adapted to use the thread-pool instead of creating threads for handling the application RPC requests.

The Mono server could assign a thread to each persistent connection. The assigned thread would be responsible for sending the updates to the client and waiting for the next getUpdates RPC request. Apart from that, however, the client thread would most probably be idling. Thus, rather than having many idle threads, we avoid the one-thread-per-client model by making the sendUpdates method responsible for not only sending an update to the client but also waiting for the next incoming
getUpdates RPC request to be buffered. This approach removes the need for allocating a dedicated thread to every client and ultimately reduces the number of active threads at the server.

The Mono server monitors the requestTimeout deadline of all buffered requests centrally. We use a priority queue (ReqTimeoutList) for this purpose and a thread is assigned to monitor this priority queue. The priority in the queue is determined by the request timeout deadline, that is getUpdates RPC requests with earlier deadlines are given higher priority. All request timeout deadlines are computed using the same requestTimeout value for all clients. Therefore, putting a getUpdates RPC request in the priority queue is as simple as appending it to the tail of the queue.

Finally, as discussed in Section 4.2, our design assumes that any RPC method execution time is minimal. Methods that require long execution time should instead return an acknowledgement message and create an auxiliary thread or use the thread-pool to continue the execution of the RPC method and asynchronously return the result to the client. Since method implementation is application specific, we leave it to the application developer’s discretion to identify such methods and include this mechanism in the application code.

5.1.2 SPEX Duo

The class diagram of our SPEX Duo server implementation is shown in Figure 5–2. As can be seen from the class diagram, we reuse most of the classes from the SPEX Mono implementation. In SPEX Duo, the server has to manage two persistent connections with each client. Thus, the Connection class is modified to hold 2
persistent connections. However, it still uses the same two-lock synchronization mechanism used in Mono.

Figure 5–2: SPEX Duo server class diagram

The Duo server implementation takes a different approach for managing the persistent connections. In Duo, we designate a thread per client to handle the client’s two persistent connections. At any single time, the client thread only needs to wait for an incoming RPC request at one of the connections as the other connection is occupied by a buffered `getUpdates` RPC request. Since there is already a thread that is responsible for receiving and processing the incoming RPC requests, both `getUpdates` RPC request and application RPC requests, the `sendUpdates` method does not need to wait for the next incoming `getUpdates` RPC request after sending...
an update. Furthermore, the load of the main server process in Duo is only to wait for login related requests as other requests are received by the client threads.

5.1.3 Authenticated XML-RPC Support

The SPEX protocol implementation has included many of the features required by the Authenticated XML-RPC scheme, for example, the login mechanism and embedding the \texttt{<qReqInfo>} in the RPC messages. As a result, to enable the Authenticated XML-RPC, we only need to add additional code for verifying the MAC of incoming RPC messages and generating the MAC of every outgoing RPC message. During the login phase, the user’s \texttt{webToken} is retrieved from the database. Once the login is completed, the \texttt{webToken} is stored as an attribute of the \texttt{Connection} instance for fast retrieval.

5.1.4 Implementation Challenges

When we first conducted our performance test, we discovered that RPC executions using HTTP/1.1 were significantly slower than executions using HTTP/1.0. We eventually found out that the additional delay was because the server’s TCP implementation had \textit{Nagle’s algorithm} \cite{22} enabled by default. When it is enabled, the TCP socket tries to combine small outgoing messages for some duration of time and sends them at once to reduce message overhead. This adversely affects the latency of the game lobby traffic as the messages of online game lobbies are typically small, frequent, and delay sensitive. Therefore, it is important to disable the Nagle’s algorithm at the socket level. Nagle’s algorithm does not affect the RPCs that use the HTTP/1.0 connection because the connection is directly terminated, flushing out the buffered message.
We also had a performance issue with the thread-pool implementation that we used. The thread-pool consists of a group of worker threads and a job queue. When the job queue is empty, the worker threads wait on a condition variable that will be signaled when a work job is submitted to the job queue. Python provides a `wait` function for this purpose. A worker thread calling the `wait` function waits on the condition variable. In the Python VM implementation we used, the use of timeouts in `wait` calls dramatically reduces performance. We solve this by not using timeouts in our implementation. The initial intent of the timeouts is for the blocked worker threads to periodically check if it should terminate by evaluating its state variable. Since the timeout is no longer an option, we use a special work job to terminate a worker thread.

### 5.2 SPEX Client

We implemented the SPEX client Java module using the Apache XML-RPC version 3 [1]. The client library of Apache XML-RPC centers around the `XmlRpcClient` class for communicating with an XML-RPC server. The client class has an `execute` method that takes care of the sending of the RPC request to the server and the unmarshalling of the returned RPC response. The `XmlRpcClient` class provides several ways to configure its operation. One configuration point that we are interested in is the flexibility of switching the underlying transport implementation, that is the HTTP client implementation. A wrapper class that implements the `XmlRpcTransport` interface is defined for each transport implementation. One of the available transport classes is the `XmlRpcLiteHttpTransport` which is the simplest
and possibly the fastest implementations in the library. However, it only supports the HTTP/1.0 protocol.

We use `XmlRpcLiteHttpTransport` to create two new transport implementations: `SpexLiteHttpTransport` and `SpexPersHttpTransport`. The first transport is an HTTP/1.0 transport for our SPEX protocol and the latter is the HTTP/1.1 version. The SPEX transport classes append the `<qReqInfo>` and `<qSecurity>` information to the outgoing RPC requests and verify the `<qReqInfo>` and `<qSecurity>` information of the received RPC responses. The `XmlRpcClient` class instantiates the transport using the `XmlRpcTransportFactory` provided. The transport factory for `SpexLiteHttpTransport`, which is `SpexLiteHttpTransportFactory`, instantiates a new transport for every RPC request/response pair. On the other hand, the `SpexPersHttpTransportFactory` only instantiates `SpexPersHttpTransport` once and uses this instance for all RPC request/response pairs. Figure 5–3 shows the class diagram of the SPEX transport classes.

Similar to the server implementation, we define `SpexMonoClient` and `SpexDuoClient` classes to encapsulate the Mono and Duo client implementations. Both classes implement the `SpexClient` interface. In reference to the SPEX framework in Section 4.3, the `SpexClient` provides the `execute` method as the implementation for the `execute-rpc` service. For the `update-callback` service, the client application provides an `UpdateCallback` instance to the `SpexClient`. The `SpexClient` uses the `UpdateCallback` instance to deliver the received server updates to the client application. The class diagram for the SPEX client is shown in Figure 5–4.
Figure 5–3: SPEX Client Transport Classes

Figure 5–4: SPEX Client Class Diagram
In our implementation, the Mono client has a background thread that loops continuously to handle the update traffic from the server and execute the application callback. The thread is blocked on the `getUpdates` RPC until an update is received from the server. When the thread receives an update, it invokes the update callback and then executes the `getUpdates` RPC again. The `SpexMonoClient` has two `XmlRpcClient` instances; one uses the `SpexLiteHttpTransport` and the other uses `SpexPersHttpTransport`. The background thread uses the XML-RPC client instance with the persistent connection exclusively for retrieving updates from the server. The other XML-RPC client instance is used only for sending application RPC requests.

Like the Mono client, the Duo client also has two instances of the XML-RPC clients, but both use the `SpexPersHttpTransport` as transport. However, the background thread that handles the `getUpdates` traffic needs to alternately use the two `XmlRpcClient` instances as have been discussed in Section 4.5.2.
CHAPTER 6
Experiments

In this chapter we present our experimental results on the scalability of our SPEX implementation described in the previous chapter. We are particularly interested in investigating the responsiveness and the scalability of the server push traffic, the new feature that SPEX brings to standard XML-RPC, of our server implementation in a game lobby scenario. In our experiments, we first analyze the performance characteristics of the SPEX server without a thread-pool, that is a new thread is created for each application RPC request. We then compare this performance with the server performance when a thread-pool is used instead. We also want to see if concurrentMcast can speed up the multicast operation. Lastly, we analyze the overhead of the Authenticated XML-RPC scheme.

6.1 Traffic Scenario

To facilitate the performance evaluation, we implemented a simple group chat service, that is Messaging.Post (refer to the example of the One-to-Many Clients interaction in Section 2.3). Each chat group consists of five players and the group membership remains static throughout an experiment. The Messaging.Post RPC request triggers the server application to multicast the posted text to every member of the group chat (including the sender himself/herself). Since the group membership is static, the text messages from a player is always multicasted to the same five players. We loaded this service by creating a hypothetical traffic where each client submits
a 50 character text message per second to the multicast service. This hypothetical traffic is somewhere in between of Instant Messaging and MMOG (refer to our discussion in Section 2.4). We assume that a player generates an event every 1 second on average in a game lobby. By multicasting to five players, we emulate the server push traffic being larger than client-originated traffic with inter-arrival time of 200 ms on average.

6.2 Methodology

We take the round-trip time (RTT) of the message posting and multicasting as a measure of the SPEX server performance. The RTT referred here is the elapsed time between the client application uses the SPEX client to send the Messaging.Post RPC request and the time it receives back its own post message through the server push channel. A graphical illustration of the RTT measurement is shown in Figure 6–1. Obviously, one of the receivers of the multicast must be the sender itself in order to measure the RTT.

![Figure 6–1: RTT measurement in the experiments](image-url)
Our RTT measurement is different from those that are normally measured at the network level. The network level RTT measures the elapsed time between a packet being sent by the server and the server receiving the acknowledgement of the reception of the packet by the client. In contrast, our RTT is measured at the application level; it includes not only the round-trip network latency, but also the SPEX overhead and the server processing time. However, we disable the Authenticated XML-RPC scheme throughout our experiments, except for the last experiment where we evaluate the overhead of the scheme.

In order to investigate the scalability of the system, we vary the number of clients starting from 20 clients up to 200 clients, increasing in steps of 20. We collected 5 rounds of data for each given number of clients. In one round, every client sends 400 messages. But only 300 out of the 400 messages are used for computing the RTT mean and standard deviation ($\sigma$). We discard the RTT measurements of the first 80 messages and the last 20 messages to compensate for the system warm-up and cool-down. The RTT mean and standard deviation (jitter) of each round is computed by combining all the 300 message RTTs of every client. The next step is to summarize the statistics from the five rounds to represent a single data point in our experimental result. The means from the five rounds can be combined by taking the average of the five values. We use the weighted variance method [11] to aggregate the five standard deviations. Since the data sets from the five rounds are uncorrelated, the formula can be simplified as follows:

$$\sigma = \frac{1}{5} \sqrt{\sum_{i=1}^{5} \sigma_i^2}$$
6.3 Test Application Implementation

We implemented a Java client application that uses our SPEX client implementation to generate the desired load. In the beginning, the client application automatically performs the login and then waits for 5 seconds before it starts sending the 400 messages to the server to make sure that all other clients have completed their logins. The client application also waits for another 20 seconds after it has finished sending the last message to receive more updates from the server.

The client application is a single threaded application. To guarantee that one message is sent per second, after the client executes the Messaging.Post RPC, it compares the current system time with the absolute deadline of the next RPC request to determine how long it should wait before sending a new request. We also modified our SPEX client implementation to record the system time when the client application invokes the execute method of SpexClient (called RTTStartTime) and the system time right before SpexClient invokes the processUpdate method of UpdateCallback (called RTTEndTime). Precisely, we can define the measured RTT as:

\[
RTT = RTTEndTime - RTTStartTime
\]

For the server application, we implemented the same Messaging.Post service for both SPEX Mono and Duo experiments. Unless specified otherwise, the Messaging.Post service uses the iterativeMcast method to multicast the text message to all five members in a chat group. Furthermore, the message sender is always put as the last receiver in the receiver list.
The *Messaging.Post* execution time will inevitably degrade as the server load increases. Since we only allow the client to open two connections to the server at one time, congested clients are prevented from submitting more requests. This limits our ability to test scalability. To overcome this issue, the *Messaging.Post* service is implemented to directly return an acknowledgment message to the client and then create a helper thread or use the thread-pool to complete the service execution. Note that, the SPEX server implementations are not modified to support our experiments.

Lastly, we set the *waitTimeout* to 3 seconds, the *requestTimeout* to 30 seconds, and the *responseTimeout* to 32 seconds. We believe these timings are reasonable for a real web-based game lobby deployment, that is the internet environment.

### 6.4 Infrastructure

Our experiments are conducted using 50 machines: 36 machines from SOCS lab-2, 13 machines from SOCS lab-3, and 1 server machine (named *rogue*). The specification of each machine in the labs is listed in Table 6–1.

<table>
<thead>
<tr>
<th></th>
<th>rogue</th>
<th>lab-2</th>
<th>lab-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon MP @ 2.70 GHz</td>
<td>Intel Core 2 Duo E8135 @ 2.4 Ghz</td>
<td>Intel Pentium D @ 2.8 GHz</td>
</tr>
<tr>
<td>No. of Cores</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB</td>
<td>2 GB</td>
<td>2 GB</td>
</tr>
<tr>
<td>Linux Kernel</td>
<td>2.6.26-2-686-bigmem</td>
<td>2.6.27-9-generic</td>
<td>2.6.27-9-generic</td>
</tr>
<tr>
<td>Java VM</td>
<td>1.6.0_0</td>
<td>1.6.0_10</td>
<td>1.6.0_10</td>
</tr>
<tr>
<td>Python VM</td>
<td>2.5</td>
<td>2.5</td>
<td>2.5</td>
</tr>
</tbody>
</table>

In our experiments, we allocated the lab-2 and lab-3 machines for running the client applications and the rogue server exclusively for the server application. Since
the number of client application instances is more than the number of client machines, a client machine may host several client application instances simultaneously with each instance running on a separate Java virtual machine (VM). Due limitations of the Python VM, even though the server is multi-threaded, it does not benefit from the multi-core processor that the server machine has because the VM only supports one running thread at any time.

Figure 6–2 depicts the network setup of the machines. The three groups of computers are located at separate locations, but they still reside in the same local area network. Every machine is connected directly to a central 100BaseT Switch with 100Mbit/s link. The network round-trip time from any machine to another machine in the network measured using ping command is approximately 0.3 ms.

![Network Setup Diagram](image)

Figure 6–2: Network setup of the machines for the experiments

6.5 Bandwidth Estimation

To estimate the bandwidth requirement of our experiment, we first captured the marshalled RPC requests and responses, including the HTTP header, to measure the length of the messages. The Messaging.Post RPC request and response messages are
646 bytes (for a payload of 50 character text message) and 309 bytes long respectively. The getUpdates RPC request and response messages are 323 bytes and 712 bytes long respectively. The measured getUpdates RPC response message carries only one 50 character text message. We can then estimate the upper bound of the bandwidth requirement at the application-level based on the length of the messages. We assume the most demanding scenario for our estimation where, in every second, the SPEX server has to forward the 5 text messages to a client using 5 getUpdates RPC responses, so there is no message aggregation. Based on this assumption, the incoming and outgoing bandwidth at the client side are 3.8 Kbyte/s and 2.2 Kbyte/s. And for the server with 200 clients, the incoming bandwidth is 441.6 Kbyte/s and the outgoing bandwidth goes up to 755.7 Kbyte/s. Although we have not taken into consideration the additional packet headers introduced by the network layers, these bandwidth requirements can be accommodated easily by the 100 Mbit/s link in our network.

6.6 Experimental Results

6.6.1 SPEX Server without Thread-Pool

We first investigate the performance of the SPEX Mono and Duo server without a thread-pool, which means the server creates a new thread when it wants to complete a task concurrently. The SPEX Mono server creates a new thread when the main server process receives the Messaging.Post RPC request and during the method execution. The SPEX Duo server only creates a new thread during the Messaging.Post execution. Figure 6–3 shows the RTT means recorded for this experiment. The servers only scale up to 140 clients in this experiment. Starting from

85
160 clients, the RTT spikes up to above 1 second for both servers and there are many instances where the server failed to create new threads. Note that, with the server’s standard thread setting of allocating 8KB of stack per thread, the Python VM can only have up to 382 threads simultaneously. Reducing the stack size to 4KB allows the number of threads to go up to approximately 780 threads; unfortunately, the same scalability problem persists. We did not try to increase the number of threads further because it is not practical to have too many active threads at the same time.

![RTT of SPEX Server without Thread-Pool](image)

**Figure 6–3:** SPEX server without thread-pool RTT measurements

We can see that the round-trip times are still well below the widely accepted threshold of 200 ms network RTT latency for real-time applications. It is important to note again that our round-trip time is a more aggressive measurement because we measure the end-to-end time at the application layer rather than at the network layer. We can also observe that Mono’s performance is clearly worse than Duo’s.
However, we do not think that this is much due to the overheads from the out-of-band connection setup of Mono’s RPC requests. To verify this, we measured the time required to open 200 TCP connections with our experiment setup. It took approximately 0.5 milliseconds on average to open a TCP connection. Thus, we suspect that most of the performance difference is due to other factors in the design of our implementation. As discussed in Section 5.1.1, the `Connection.sendUpdate` method in Mono not only sends an update to the client but also waits for the next `getUpdates` RPC request. This wait must be done for all the 4 other clients before the fifth message can be sent to the final receiver where the RTT is measured. To support our hypothesis that the performance difference is due to the `Connection.sendUpdate` implementation in Mono, we create a version of the Mono server, `no-wait Mono`, wherein the `Connection.sendUpdate` method creates another thread to wait for the next `getUpdates` RPC request. The performance of this Mono server is recorded in Figure 6–4. We can see that when the number of clients is small, the performance of the no-wait Mono server is almost similar to that of Duo, an improvement over result on the original Mono. This is consistent with our hypothesis. Disappointingly, the new Mono server performance becomes closer to the original Mono as the number of clients increases. Here, thread management overhead becomes a bottleneck, as the no-wait version requires six times more threads.

The standard deviation of the RTT, as a measure of the jitter, is recorded in Figure 6–5 as a percentage of the RTT mean. The percentage ranges from 20% to 40%. The number may seem rather high, that is roughly double of that reported in [7]. However, our measured RTT is also subjected to the process and thread
Figure 6–4: SPEX no-wait Mono server without thread-pool RTT measurements scheduling at the application level and also the sending of messages to 4 other clients. Furthermore, our RTT mean is considerably small to begin with because of the negligible network latency. As the network latency features more prominently in the RTT, the percentage of jitter would become smaller. For the rest of our experiments, we also observe the same range of RTT jitter, that is between 20% and 40%. Hence, we are not going to discuss the RTT jitter measurements again in the subsequent sections. Note that, although the standard deviation for Duo appears higher than Mono in Figure 6–5, it is not consistently the case in our experiments.

6.6.2 SPEX Server with Thread-Pool

In this section, we evaluate the performance of the SPEX server using a thread-pool instead of always creating new threads. Figure 6–6 shows server performance with 15 worker threads. In this configuration, the Mono server can only scale up to
120 clients. After this point, the thread-pool job queue just keeps growing and the CPU utilization goes above 90% as observed using the `top` Linux command. The Duo server performs slightly better as it scales up to 140 clients. Until 60 clients, the performance of the Mono server is similar to the threaded version. As the number of client increases, the performance of the Mono server with thread-pool drops by about 10% relative to the threaded server. The Duo server appears to benefit from the thread-pool when there are less than 100 clients where the performance improves by 3% to 8%. After that, the thread-pool server performance deteriorates by 10% to 40%.

We also investigated the effect of increasing the number of worker threads in the thread-pool. Figure 6–7 shows the performance of the servers with a thread-pool of 50 threads. For the Mono server, the performance is worse by almost 10%
compared to that with 15 threads thread-pool for small a number of clients, that is, up to 80 clients. However, with 50 threads, the Mono server can now handle up to 140 clients. Significant performance gains of 20% and 25% are also recorded for 100 clients and 120 clients respectively, and the same tendency can be seen in the Duo’s performance, albeit to a smaller degree. The server performance with 50 threads is marginally worse than with 15 threads for 80 clients and less. Although not as dramatic as in Mono, the RTT measurements for 100 clients to 140 clients are improved by 5% to 20%.

6.6.3 SPEX Server Using concurrentMcast

In this experiment, we investigate the behavior of the SPEX server if the concurrentMcast service is used instead of the iterative one. We conduct this experiment using the SPEX server with a thread-pool of 50 threads. The performance of the SPEX server with this configuration is recorded in Figure 6–8. The Duo server can only go up to 140 clients in this experiment. Interestingly, the Mono server can support up to 200 clients, although, realistically, having an RTT larger than 1 second is unacceptable.

In Figure 6–9, we compare the performance of the server using concurrentMcast with the one using iterativeMcast from the previous section. The Mono server with concurrentMcast exhibits huge performance improvements for a small number of clients. The main reason for this is because the SPEX server can send the message to the next client while waiting for a new getUpdates RPC request from the client. However, the performance gain drops as the server serves more clients and, after 80 clients, sending the messages concurrently actually makes the server
Figure 6–6: SPEX server with a thread-pool of 15 threads RTT measurements

Figure 6–7: SPEX server with a thread-pool of 50 threads RTT measurements

91
Figure 6–8: SPEX server using \texttt{concurrentMcast} RTT measurements

(a) Measurements for 20 - 140 clients

(b) Mono Measurements for 140 - 200 clients
perform worse. It seems that the overhead of having many smaller work-jobs becomes too high eventually. In the case of Duo, concurrently sending the 5 messages drops the server efficiency rather significantly. This is caused by two factors. First, the `Connection.sendUpdate` method in Duo does not make the executing thread wait for the client’s next `getUpdates` RPC request, thus it can start sending to the next client immediately. Second, the Python VM only allows one thread running at a time although the CPU can support multiple threads running in parallel. Basically, the `concurrentMcast` is just introducing unnecessary overhead into thread-pool work-job management and causing excessive thread switching in this case.

![concurrentMcast Performance Gain](image)

Figure 6–9: `concurrentMcast` and `iterativeMcast` performance comparison

### 6.6.4 Authenticated XML-RPC

Lastly, we want to see the performance impact of the Authenticated XML-RPC scheme. In Figure 6–10, we compare the RTT measurements of the SPEX Duo server...
without a thread-pool with the Authentication XML-RPC scheme enabled and disabled. For 20 clients, the overhead of the authentication scheme is 10%, rising to 76% for 140 clients. The sudden jump at 140 clients is because the server CPU utilization is already almost at the maximum and it is aggravated by the authentication operation which demands CPU time. Despite the jump at high loads, in comparison to HTTPS which burdens the server greatly (refer to our previous discussion in Section 3.1), the Authenticated XML-RPC scheme performance impact is considerably lower.

Figure 6–10: Threaded SPEX server with Authenticated XML-RPC RTT measurements

6.7 Summary

Our SPEX server implementation can scale up to 140 clients in our experimental setup while still offering good performance, that is below 100 ms of round-trip time. The Duo consistently performs better than the SPEX Mono throughout our
experiments. The SPEX server performance can be improved by using a thread-pool. However, having too few or too many threads in the thread-pool degrades the SPEX server performance. Therefore, the projected number of clients should be known in determining the thread-pool size. Within our Mono client design, we observe that the concurrentMcast performs better than the iterativeMcast. In contrast, the concurrentMcast impacts the Duo server performance adversely. Lastly, the Authenticated XML-RPC scheme can be seen to have low overhead as long as overall server load is low. Server performance deteriorates faster when it has high load if the authentication scheme is activated, because the authentication operation exhausts the server CPU capacity faster.
CHAPTER 7
Conclusion and Future Works

The web has become a platform of choice for online applications. Nowadays, the web is able to deliver rich content to end users on demand. The web traffic is also the least blocked traffic in the Internet. This makes it easier to reach most of the end users. However, implementing a web-based game lobby is not straight-forward. One major obstacle is designing the communication middleware for the game lobby using web-service technology and HTTP. The request-response paradigm of HTTP does not support the bidirectional nature of typical online game lobby traffic. In this thesis, we presented SPEX as a solution to this problem by introducing the long-polling technique from BOSH into the XML-RPC framework. Since the human readable XML messages of XML-RPC are not secure enough for game lobby purposes, we also introduced the Authenticated XML-RPC scheme to provide authentication guarantees on the RPC messages. Finally, we have successfully implemented a web-based communication middleware for a game lobby using our proposed protocols and conducted a performance evaluation.

Our experimental results using a simulated lobby traffic show that, with our server configuration, the implemented protocol can support up to 140 clients while still providing a good response time that is acceptable for real-time interaction. This demonstrates that our protocol can be applied for lobby implementation. Actually, if we use Quazal’s statistics, where each client roughly sends 5 messages per minute
(our experiments used a rate of 60 messages per minute), then we can even support 12 times more clients, that is 1680 clients. We have also verified that the Authenticated XML-RPC scheme only introduces a small performance overhead.

As the next step, we will implement a fully functional web-based game lobby using our web-based communication middleware. We would also like to see how we can optimize our implementation to support even larger numbers of clients. Of course to really scale up to millions of users, we would have to investigate alternate (distributed) server designs. Another issue that we would like to investigate is the communication reliability. The SPEX framework does not have any mechanism yet to recover from communication failures.
References


[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.


