Design and Implementation of a Distributed Todo List System (Codename Yoshi)

A Project for Harvard University CS262

Michael Tingley[*], Rafic Melhem[†], Randy Miller[‡], Kyle Solan[§]
Michael Traver[¶], Zhuo Yang[‖], Tyler Zou[**]

4 May 2012

1 Inspiration

1.1

We were inspired by one of Paul Graham's startup ideas essays, the full text of which is located at `http://paulgraham.com/ambitious.html`. The small segment below gives the general idea of the system:

> Email was not designed to be used the way we use it now. Email is not a messaging protocol. It's a todo list. Or rather, my inbox is a todo list, and email is the way things get onto it. But it is a disastrously bad todo list.

Based on this idea, we sought to design and build a messaging protocol more suited to todo list tasks than is our current text-based email system.

1.2 Grapevine

As far as the context of Computer Science 262 goes, the Grapevine system was an inspirational system that guided the initial design of the Yoshi system. Grapevine had an excellent model for our system – at a high level, it was a consistent and distributed mail system. It

[*]Harvard University
[†]Harvard University
[‡]Harvard University
[§]Harvard University
[¶]Harvard University
[‖]Harvard University
[**]Harvard University

was also basic enough that we could understand the essential nature of the system without a large amount of additional research.

Our Yoshi system uses registration and messaging servers to transmit and persist information. The messaging servers are designed with slightly more responsibility in the sense that they must also synchronize information between accounts. We also designated the message servers to be fully replicating — that is, each message server has messaging information for all accounts. This is perhaps less than optimal: as more users sign up for the system and send messages, the system will become increasingly unable to store all of the files. However, if given enough time to design an intelligent and robust distribution function for messages over various servers, we could eliminate this problem. In one light, this is perhaps trading scalability for consistency.

2 Design Goals

2.1 CAP Tradeoffs

Our system is designed to achieve availability, partition tolerance, and eventual consistency. The algorithm we use to achieve eventual consistency is discussed in section 5.

2.2 Practicality Concerns

Practical concerns include things like permissions for who can send tasks to you, how they can set the priority, who can edit your tasks, etc. We did not address these concerns in our system due to the scope of the project, but a final implementation would need to resolve these.

We adapted our project for a particular audience: a corporation of medium to large size. The goal would be to deploy our system internally so that security is not the responsibility of our system. This translates to the assumptions that users follow the guidelines for use of the system and do not try to maliciously manipulate the system. For example, our project does not set permissions on who can send a particular user tasks, who can edit which tasks, who can set the priority of tasks and how, and much more. If mistakes like these are avoided most of the time by most of the userbase, the our project should work as it was intended to work.

Additionally, our project is meant to handle scale of audiences this large; it is not meant to handle millions of requests per second or replace large email services like Gmail for instance. The authors of the Grapevince system made these assumptions as well, as they noted that their system did not perform well as when the userbase swelled past the authors' expectations. Similarly, we expect our system to only handle a userbase of the target audience size well.

3 Overview and Control Flow

Our project consists of four components: clients, a load balancer, message servers, and a registartion server. Generally, clients send messages to the load balancer, which forwards messages to a message server, which then ask the registration server to authenticate the identity of the client before properly handling the request. We examine each one of these stages in greater detail in the following sections:

3.1 Client

A client can send messages in one of two forms: a task to be sent to other accounts, or a request to be updated with the newest messages that were sent to the account of the particular client. The client sends all messages to the load balancer, regardless of request type. To send a task, the client simply converts a task into a binary format and sends it across a socket to the load balancer, and then opens another socket to listen for an acknowledgement from a message server. For retrieving tasks, the client sends a request with the timestamp of the latest message that it has received over a socket to the load balancer, and then opens another socket to listen for a list of new tasks from a message server.

3.2 Load Balancer

The load balancer forwards messages received from the client to a message server. The load balancer determines which message server should be sent a particular message according to a few rules. First, in order to promote per-account consistency, an account's message should be sent to the same message server, rather than depend on the message server system to synchronize messages and use more expensive algorithms like Paxos. Second, if an account has not been assigned a message server, or the dedicated message server for that account has failed, then the load balancer reassigns the account to be sent to the least-loaded message server permanently, until that message server fails or the account is removed.

The load balancer also handles bootstrapping for the servers. Because the load balancer maintains a list of message servers, a new message server or a new primary registration server can request a list of all message servers.

3.3 Message Server

When a message server receives a message from a client, it immediately extracts the account and password information from the request and requests the registration server to authenticate the account information. If the authentication fails, then the message server ignores the request. Otherwise, the message server handles the request according to its content. If the client has requested the message server to retrieve the latest tasks after a given timestamp, then the message server composes a response with the appropriate arguments and responds directly to the client. If the client has requested to send a task to another account, then the message server propagates the task to other message servers. The receiving client receives the message by executing a retrieve request. It is up to the client to determine how often it should retrieve messages.

Besides internal messages, the only other message a message server sends is at start up, at which it asks the load balancer for a list of all message servers from which to replicate account data as well as the address of the primary registration server.

3.4 Registration Server

The registration server only receives messages from the message server, and those messages are only of one type: a request to authenticate a particular account with a given password. If the registration server determines that the password does not match for a the given account, it returns a message with a false boolean indicating that authentication failed; if the password does match, then the registration server sends the message server a message with a true boolean.

The registration server is implemented in a master-slave configuration, meaning that there is only one registration server (a primary registration server) that handles authentication requests. At start up, the primary registration server simultaneously tells tells the load balancer that it is the primary registration server and asks the load balancer for a list of message servers so that it can broadcast to those servers that it is the master. This statement and broadcast also occurs when a slave assumes the role of master.

4 Implementation

4.1 Overall Structure

As stated in section 1.2, the structure of our project was inspired by the Grapevine distributed system. As such, the three main types of systems comprising the systems are message servers, registration servers, and clients. There is also a load balancer to help delegate tasks and prevent any single server from having to process a relatively excessive amount of information. Collectively, these types of systems constitute the working Yoshi system. A user account can be originated at a client and registered at a registration server. The client can then send messages to the Yoshi system. The messages will be transparently forwarded to the message servers and synchronized between them as appropriate.

4.2 Message Server

The message server is responsible for storing messages for each account and maintaining consistency within the system to ensure that these messages can be readily accessed by a client.

4.2.1 Creating a New Server

At a high level, creating a new messaging server was designed to be a simple process that requires no input from the user. The process of creating a new server would cause the server to attempt to register itself with the load balancer and then subsequently attempt to establish the necessary connections within the system to use the server as a viable message server. This is outlined in more detail in the next section.

4.2.2 Initialization

Initialization of a new messaging server is done in such a manner as to ensure that the new message server is *eventually consistent* with regards to the current messages logged by the system.

Loading or reloading the database.
The first step when initializing a message sever is to attempt to load a database. This means one of two things. Either this message server is a new message server, or this message server was a previous message server that crashed and is now being rebooted. In the event that this is a new server, this server will create a new database and bind this messaging server with a new, unique identifier. In the case of the latter, the server will load the database and its messages and will attempt to "catch up" with the rest of the system by polling neighbor messaging servers (covered later).

Connecting with the Load Balancer.
Communicating with the load balancer is a low-bandwidth, concentrated way that the messaging server is able to gather all necessary information from the rest of the system. By sending a request to the load-balancer, the message server is able to retrieve a list of both registration servers, and a partial list of message servers. By the design of the system, this process is transparent—the load balancer could either directly return the results to these queries, or it could return petition a registration server to return the relevant lists.

Either way, once the message server has a list of registration servers and a (partial) list of message servers, it can attempt to synchronize itself with the rest of the system. In the previous process (communicating with the load balancer), the message server effectively registered itself with the registration servers. However, it has not yet registered with the message servers or synchronized its list of messages.

Registering and Synchronizing with Message Servers.
We have bundled the two processes in this system into one for efficiency. The system will "ping" all known message servers (which may, in fact, not be all message servers in the system, but rather a known subset). This causes these message servers to acknowledge the new system in the chain. This new system is then registered to receive updates from these servers whenever they receive new messages or updates of old ones.

In addition to simply registering this server with other neighboring message servers, this process will cause the nearby servers to send a list of update messages to this server. This server will send a symbolic identifier (namely, the totally-ordered message ID number for all known accounts) to all nearby message servers. These servers will respond with a list of "newer" messages (assuming there are any). Thus, this message server will be eventually consistent in the system. For more information on this process, see section 5.

4.2.3 Handling Messages

We handle messages in two different ways, depending on whether the message is a request or a response. Requests are handled asynchronously by a dedicated thread. We spin off a new thread for each client connection, and we spin off a new request handler thread for each request received. This maintains the availability of the message server to accept new clients and requests. If the message is a response, we handle it synchronously inline. That is, we wait for a response after sending off a request, in much the same way as a conventional method call waits for a return value. The response is then processed synchronously as a method return value would be.

However, there are some cases in which we expect no response from a request, so we don't wait; this is similar to a method call with a `void` return type. An example of this is `PING_MS` – we alert other message servers of our existence as a message server, but we expect no acknowledgment or return value to to be sent back.

4.2.4 Database

We use pair a MongoDB database to store the task objects associated with messages. MongoDB is an open source, document oriented, NoSQL database, which we chose to use mainly because of its ease of installation. In a distributed system with many message sever machines a database that requires a single binary and a single data directory is a godsend that take five minutes to install. Additionally, our message server code is in Java, so the method based query and storage of MongoDB was very easy to learn. The database maintains its own identity in a collection separate from the tasks, so that new message sever instances can be started, connected to the database, and continue operation without data loss.

4.3 Registration Server

The registration server is responsible for maintaining account information for each user of the system. This information is stored as a (name, password) pair in a replicated SQL database, and the registration server is set up with a primary-backup scheme.

4.3.1 Main Functions

The registration server supports the following two main functions:

1. CreateAccount(name, password) – creates an account with the given name and password, storing the information in the SQL database
2. Authenticate(name, password) – authenticates the given name and password, returning either `true` or `false`

4.3.2 Database Replication

The replication method is a primary-backup approach: the primary registration server is responsible for all requests directed to registration servers, and if the primary fails, the next backup registration server replaces it. To ensure that the backup has the most up-to-date information when it replaces the failed primary, all backup registration servers retrieve updates from the current primary as they are made. The MySQL replication is done more or less independently from the other services. Fortunately for us, MySQL provides easy replication functionality that sets one server as a master (primary) and others as slaves (backups). In order to achieve replication, slaves read a log produced by the master, executing new updates on their databases.

Of course, the latency induced in this replication process makes it an eventually consistent system. However, in all of our tests, the replication is fast enough that the latency is non-detectable by humans.

4.3.3 Failure Detection

One registration server is started as the primary server and all others are started as backup servers. To check for failure of the primary, a “heartbeat” ping message is sent at constant intervals from the backups to the primary. Once the primary receives the ping message, it sends a short “still alive” response.

Once the primary fails, a backup will be chosen randomly from the list of current backups to take over as the new primary. It will poll the load balancer for the list of message servers and send a message with the IP of the new primary registration server. The other backups will update their configurations accordingly to replicate the new primary server.

4.4 Load Balancer

The load balancer for this tasks system is not a load balancer in the traditional sense. In this case, the load balancer is the centralized information store of the system, as it is responsible for maintaining a list of all known message servers, a reference to the primary registration server, and for redirecting clients requests to the appropriate message server, determined by the client's account number. Thus, it has the following interface to clients:

1. Send Task(sender account, password, recipients, task)
2. Retrieve Tasks(sender account, password)

and the following interface to message and registration servers:

1. Retrieve a list of all message servers()
2. Retrieve the address of the primary registration server()
3. Set the load balancer's reference to the primary registration server() - sent from the primary registration server
4. Destroy message server(server id)

We now consider the reasons behind why the load balancer supports this interface as well as how it is implemented.

4.4.1 Motivation

The load balancer supports this interface primarily because of the design decision to have each account be dedicated to a single message server, even though all accounts are on all message servers. Because accounts have dedicated message servers, message of clients of a particular account must be routed to the corresponding message server; however, new clients do not know which message server is the appropriate one to send requests to. Rather than have all of the message servers try to maintain and synchronize a list of all servers, it is simpler to have a central point of contact for a list of all servers. This has the added benefit of increased availability and better response times for clients, since messages go directly to the dedicated message server and can get a response immediately. Additionally, as discussed in the Message Server section, having a dedicated message server per account means that an account is at least eventually consistent and is sometimes consistent relatively quickly per account. This property makes retrieving tasks relatively simple for the message servers, and also promotes a faster response to a task retrieval request. Thus, the load balancer's interface is necessary in order to forward the client's request to its dedicated message server.

As mentioned before briefly, maintaining a list of all message servers in one central point of contact greatly simplifies the complexity of message servers. Maintaining a comprehensive, consistent list among a distributed system of servers could have been implemented using the Paxos algorithm as examined in class, but we felt that the message servers were complex enough. So, we decided to use the load balancer as the keeper of the message server list. As discussed above, this also easily allows clients' messages to be rerouted to the correct server.

Since the load balancer already serves as a central hub for the message server list, it makes sense to also keep the address of the primary registration server in the load balancer as well. Thus, if a message server does not know what the address is of the primary registration server, instead of asking a peer and risking a out-of-date address, it can instead query the load balancer for the address and be immediately confident that the address is correct. Again, we could implement a message server-polling algorithm here for this request, but since the message server queries the load balancer for a list of message servers anyway, it makes sense to get the primary registration server from there as well.

Because the load balancer has these two pieces of information, it serves as the primary means for bootstrapping servers. Each server has a config file at startup that contains the address of the load balancer. New message servers can then query the load balancer for a list of all message servers in order to connect itself to the message server graph. New registration servers can also query the load balancer for the current primary registration server to include itself as a slave in that system as well. Our system assumes that the load balancer will bootstrap itself before all other servers go live, followed by the primary registration server, followed by at least one message server, follwed by the clients and any more message servers.

4.4.2 Implementation

The load balancer is ultimately a dedicated server that spawns background workers to handle requests. At bootup, the load balancer initializes an empty 'store', which keeps track of accounts, servers, and the relationships between the two types of objects. Then, it opens up a socket to listen for connections from clients and servers. Once it receives a connection, it spawns a background worker to read data over the connection and ultimately handle the request.

Once it reads in data over the socket from the requester and parses that data into a message, each background worker handles each request according to the opcode of the request, and ignores requests that are not supported in the load balancer interface. The background worker either forwards the message to a message server (determined by the account number of the client sending the request) or responds to the query after consulting the store.

The load balancer has a single, thread-safe, and concurrent data stucture (called the store) for keeping track of servers as well as the account-to-message-server mappings. It is optimized to provide for quick fetch operations which include random fetching and querying for all servers. Because there is no built-in Java structure that supports these reads and is thread-safe, we created a concurrent wrapper class to support the load balancer's needs. Additionally, there are sub-data-structures used to greedily redistribute accounts when a message server dies, destroyed, or removed from the message server graph. More detail on this implementation is available in the Javadoc for the load balancer.

4.5 Client

The client consists of two components: the actual client, and a webclient server responsible for communication with the load balancer. The webclient server has the following client-facing interface:

1. send(account, password, recipient, task)
2. retrieve(account, password, virtual time stamp of latest received message)

When the client calls one of the above methods, the client's server translates the python objects into serial data using the Python version of Google's Protocol Buffers, sends the data to the load balancer, and listens for a response from a message server.

4.5.1 Client

The client itself is an end-user machine that accesses the Python-based webclient server. Users log in and use the webclient as a standard website, with the webclient server seamlessly taking care of all the distributed details, so it appears as a single cohesive service to users.

4.5.2 Webclient Server

The most direct client of our service is also a server itself. In this case, our client is a web server that has the ability to support multiple simultaneous users through a single client interface.

The web server is built on Django, a Python-based web framework. We chose this particular framework because it interfaced well with our portable Python client library. Our server simply uses the library's send() call to send messages to message servers, and retrieve() calls to poll message servers for new messages. It caches received messages in a MySQL database so that users do not have to reload every message from the message server backend whenever they access the client. The webserver asks the message server for any messages with a virtual timestamp value is greater than the largest received timestamp value, stores them in its database, and then serves users messages out of the database.

Similar to modern web email, multiple iterations of these webclient servers can exist, and can provide a different look and feel to the user or serving different subsets of user accounts while still ultimately using the same distributed backend.

4.5.3 Communications Library

As mentioned, the client's server communicates with the load balancer via a library. This abstracts most of the difficult work away from the client, so they don't have to worry about finding the load balancer, converting task objects into serialized protocol buffers, and sending them over a socket. It's highly portable, and supports a variety of client implementations: anything from a command-line script to an Outlook-style email client. The library offers the client a uniform object representation of a task, which can be passed to the library's methods to be sent as serial data to the load balancer and message server. For the send method, the library is responsible for returning a boolean from a message server response indicating that the message server successfully received the message. For the retrieve method, the library is responsible for decoding the message server's serial response into a task object array, which as mentioned previously is defined by Task.py in the library.

4.6 Wire Protocol

We use Google Protocol Buffers to implement our wire protocol (`https://developers.google.com/protocol-buffers/`). Protocol buffers provide a convenient and efficient way to pass messages over the network. Indeed, protocol buffers are an "automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler"[1]. Google provides a simple language for defining message types, and the protocol buffers library automatically generates code for building and parsing messages. We implemented an RPC-like set of opcodes using this system.

4.7 Opcodes

The following list of operation codes represent the entire interface among machines in our distributed system.

1. Ask the load balancer for the IP address of the primary registration server.

    QUERY_LB_FOR_RS = 1.

[1]`https://developers.google.com/protocol-buffers/docs/overview`

A message server requests the load balancer for the address of the primary registration server when it bootstraps.

2. Ask the load balancer for the IP addresses of all message servers in the message server graph.

   QUERY_LB_FOR_MS = 2

   A message server requests the load balancer for the address of all message servers when it bootstraps in order to insert it itself into the message server graph and obtain account data.

3. Alert message servers of a new message server's existence, sent from the new message server.

   PING_MS = 3

   A new message server inserts itself into the message server graph by informing other message servers of its presence.

4. Send a task to a message server from a message server.

   TASK_TO_MS = 4

   This operation code is used when a message server wants to propagate a task it has received to other message servers.

5. Poll a message server for all messages for a given account, sent from a new message server.

   POLL_MS = 5

   A message server requests any new data from a message server at both start up and periodically during normal execution in order to have fresh information.

6. Poll a message server for all messages for a given account, sent from a client.

   CLIENT_POLL_MS = 6

   This is used when a client wants to retrieve new messages that have been sent to its account.

7. Send a task to a message server from a client.

   TASK_FROM_CLIENT = 7

   This is used when a client wants to share a task with another account or client.

8. Authenticate a client, sent from a message server.

   AUTH_CLIENT = 8

   This is used when a message server receives a task from a client and wants to authenticate the identity of the sender of the request.

9. Broadcast out to all servers that there is a new primary registration server

   NEW_PRIM_REG = 9

   This operation code is used when a primary registration server starts up, both initially and when a backup replaces an old primary registration server.

10. Open a new client, sent from a client

    NEW_ACCT = 10

    This opcode is currently not used, but could and should be supported in future versions.

5 Achieving Eventual Consistency

This section will discuss various protocols used in the system that ensure that the system will achieve eventual consistency. (In fact, if we assume a reasonably fast transfer rate of messages across the wire, this eventual consistency should be achieved in a very short time span.)

5.1 Adding a New Server

Regardless of the type of server used, we employed an algorithm when incorporating a new server that would ensure that the system would necessarily achieve eventual consistency. If a server X is incorporated into the system, it registers itself with a set of other servers $S_1, ..., S_n$ (if it fails to register with *any* of them, then this means that an insufficient number of systems are live in the Yoshi System to guarantee systemwide-consistency). Then, we have two scenarios. Either $\exists S_i \in S_1, ..., S_n$ that is consistent, in which case when X requests an update from S_i, it will become systemwide consistent. Note that this works because we have a totally-ordered protocol, and therefore the system will know which messages and states are absolutely more recent than which others (this is discussed later in this section).

The alternative case occurs when $\forall S_i \in S_1, ..., S_n, S_i$ is not absolutely consistent. Well, in this case, we know that $\exists S_j \notin S_1, ..., S_n$ that is absolutely consistent (otherwise our original system would have been consistent). Then, there must be some chain of mutually-registered servers $S_{i_1}, S_{i_2}, ..., S_j$ for some $S_{i_1} \in S_1, ..., S_n$ (otherwise the system would not be connected). Then, since S_j is absolutely consistent, *eventually* it will forward all messages successfully to its mutually-registered servers (this is how the algorithm is designed). Therefore, *eventually*, messages must be forwarded from S_j to X, making X consistent.

We see then that, in either of these scenarios, the system will necessarily achieve eventual consistency.

5.2 Total Ordering

An important premise of our system is the ability to achieve a total ordering on messages *per account.* As we have discussed in class, total ordering is difficult to achieve—it is often done at the expense of availability. Here, too, we make a trade-off—however, it is availability on a *per account* basis rather than a full system basis. It is expected that per-account consistency represents a small subset of the entire system, and would be achieved in a small time-frame. Furthermore, if we assume a message transfer time of β between servers, we can guarantee a per-account consistency time of $\beta \log(n)$, where n is the number of servers, if we assume a server graph in which each server has approximately the same number of outgoing edges (≥ 2). It is expected that, in any reasonable modern system, β would be small.[2]

Then, how do we go about achieving this total ordering? In the load balancer, we ensure that clients of the same accounts are sending messages through the same (live) message servers. Then, whenever we send out a message, we bind to it a unique and incrementing integer "timestamp" value (similar to Paxos). This message gets forwarded to the rest of the system in a DFS-style message send. That is, the root message server would forward the message to all neighboring severs. It would refuse to send a message *from the same account* until after it received an "accounted for" message from the servers that it forwarded the message to. However, these servers would also forward it to their neighbors, and would themselves not send an "accounted for" message until all of their neighbors had sent the same message to them. Whenever a server tries to send a message to another server that already knows about this message, the server that already knows about the message would automatically send an "accounted for" message back. Thus, an "accounted for" message would be sent on any of forwards and backwards edges. Then, we have that the algorithm must terminate. The described algorithm is essentially a distributed implementation of a Depth-First Propagation, and so we know that (1) live all servers will receive the update and (2) the root server will not report "accounted for" until all other reachable servers know of the update.

This process is carried out for the first message for all accounts in addition to all subsequent messages. Then, by a simple inductive argument, we know that a server must be consistent for an account before it will receive an additional message for that account. Therefore, the process ensures a total ordering that is both consistent and bounded by a reasonable time factor.

[2]In our implementation, we allowed β to be upper-bounded by 30 seconds, and caused a time-out if this was exceeded, treating the server as "down".

6 Conclusions

6.1 Results and Conclusions

In large part, we deem the program a success. Although it did not work as optimally as we had originally hoped, as Professor James H. Waldo says, “Distributed Systems Are Hard.” Working with a 7 man team was a challenge indeed, and coordination was challenging. But we’ll save that for the next section.

The program was able to synchronize message between accounts and for the most part ensure that messages were consistently delivered to clients. Messages were successfully transmitted between message servers, and the database was able to be synchronized for multiple clients on different physical computers. Messages could be sent from one client to another, synchronized, and updated.

We were indeed able to synthetically crash a server and bring it back up under another IP address. The device was able to be re-integrated with the system and could be used to send and store messages for other clients as well. We believe that we have a robust and reasonably extensible system that could be expanded to a context of several thousand devices very easily. Expanding to the order of hundreds of thousands of devices may be challenging due to the disk space management issues in the system. However, given enough time, we would be able to discern a new and more extensible algorithm that would allow us to distribute the resources of the system more effectively.

6.2 Challenges and What We Learned

A major—probably *the* major difficulty—in creating this system was in coordinating with 7 team members. Management was a challenge, and creating protocols and paradigms that were compatible was even more difficult. We attempted to make interfaces that allowed for autonomous development between individual teams; however, since message-level communication was an integral part of this system, it was very important to have consistent messages over the wire that were comprehensible to all servers involved. Even though we used Google’s Protocol Buffer semantics when sending messages, keeping this consistent between team members turned out to be truly difficult, given the imperfect communication when designing wire protocols. This resulted in an amount of code having to be rewritten, and compromises having to be reached between the registration and messaging teams.

Google’s Protocol Buffers also turned out not to be compatible across different languages. Specifically, the client (implemented in Python) sent messages to the load balancer over a socket and the data was automatically delimited, so the load balancer would parse the message using a standard method call. However, data from the servers (implemented in Java) were not delimited over the sockets, so another method was needed to delimit and then send the data. Ultimately, this meant that we needed to manually delimit our messages, which turned out to be more of a nuisance than we originally thought.

One major challenge was designing the system and determining which component of the system was responsible for what parts. Originally, the registration servers were to serve as the central point of contact for the system, but because the information needed to be globally consistent, and since the registration servers were distributed and not fully consistent, we realized we had to put the information somewhere else in order to focus on getting the other components working first. If we had more time, we might have attempted to relieve the load balancer of its duties of storing the global information and place that responsibility on another component or perhaps a new, dedicated component.

Another significant challenge was, of course, testing. Configuring the servers was very difficult. Although getting the servers working on our local machines was (challenging but) doable, getting them set up on remote servers was a multi-hour ordeal that forced us to deal with restrictions of the FAS servers, and saw many restarting of servers.

Deciphering some of the Protocol Buffer API functions was also challenging, and required extensive trial-and-error.

7 Future Work

There are many features of the Yoshi specification that could be enhanced with additional research, implementation, and testing. Most imminently, we would have perhaps redesigned part of the system to ensure that messages were lost under no circumstances. We would also build in a more robust system for handling multiple recipients.

One of the main specifications that would need to be enhanced to permit scalability is to fix the data distribution shortcomings that we had with our system. We need to design a robust and consistent way of distributing messages across a large number of message servers. Currently, all messages are stored on all servers. Obviously, this is not good practice in terms of scalability. However, designing a fix to this is extremely non-trivial, and would have significant implications in terms of consistency (and to some extent availability).

The client could also be expanded to be much more feature-rich, although that was, for the most part, out of the scope of this project. Because Yoshi describes a complete protocol, it should be a theoretically "simple" task to allow for third-party development of clients and plug them in to the system.

We're also not sure what kinds of problems the Harvard network was hiding. During testing, we were all using the Harvard network, and in many cases were VPN'd directly into the network. However, if we expanded the protocol to arbitrary machines, we may have encountered other problems that were not accounted for in this project, such as in terms of firewalls and incompatible permission rights. We were also able to debug our system directly and with a group of other highly-knowledgeable technicians (i.e. the members of our team), so in a real-world implementation, support would need to be of a much greater priority than it was for this project.