Fast Flux Chat

Greg Banks, Nick Childers, Sean Ford
Department of Computer Science
University of California Santa Barbara
nomed@cs.ucsb.edu, voltaire@cs.ucsb.edu, sford@umail.ucsb.edu

June 9, 2008

Abstract

In this report, we present a design and implementation of a robust chat network heavily influenced by the popular IRC protocol. By utilizing the techniques underlying Fast Flux networks and making use of onion routing, our network is extremely difficult to track and take down. This behavior is implemented as a simple extension to the Compute Farm network topology allowing us to easily control network resources from a protected central authority.

1 Introduction

One of the primary inspiration for our Fast Flux Chat Network comes from the well established and popular IRC protocol. A large part of IRC's success comes from its simplicity and scalability. However, security is not a major factor in the design of IRC, as clients are easily identifiable and encryption is non-existent. By extending the basic foundations of the IRC protocol with additional security features, the Fast Flux Chat network is both extremely hard to take down and provides greater protection and anonymity to the end user.

Our primary goal in designing this network was to ensure that it was robust against failure, accidental or otherwise. Specifically, we wanted to make it difficult for a determined attacker to bring down the network by disabling key infrastructure. Our second goal was to have an ethereal network presence in the sense that the majority of the network components have a very short lifespan and can be disabled with little effect to the network as a whole. Central to these goals is the idea of a Fast-Flux network that make use of disposable components. By implementing this Fast Flux behavior as an extension to the Compute Farm network, we were able to meet these goals and develop a more robust and secure chat network.

2 Related Work

There has been much work done in the areas of onion routing [5] and botnet detection. Our work is concerned with two aspects, or results from these areas. First, our chat network makes use of the botnet related idea of "Fast-Flux Service Networks" [7]. These networks, which are basically just a large number of bots being used as reverse proxies for malicious websites, have become quite popular over the past two years. They have been recognized, although not named as such, as far back as 2003 when it was observed that a particular spammer's scam website seemed to be migrating very rapidly among many different hosts [1]. At present, there is much work being done to both detect these Fast-Flux service networks as well as to utilize their inherent characteristics in order to discover and categorize various botnets [6]. The possible uses of these networks, however, has not been explored or elaborated on. We hope to provide an example of one possible application with our Fast-Flux chat network.

Onion routing is a second area of study that is related to our tool. This study has culminated in the readily available and practical Tor network [2]. This is a network that provides end-users with access to a cloud of onion routers and the ability to build circuits through this cloud. The network provides the ability to navigate the Internet with a high degree of anonymity. Several services have been built on top of Tor. One such service that is closely related to ours is Scatterchat [3]. This chat network routes all of its traffic through the Tor network and provides end-to-end encryption, thus providing its users with a high degree of privacy and anonymity. One problem with this network, however, is speed. We aim to provide a similar chat service but with the focus on the network itself not being taken down. We therefor utilize Tor to hide critical components and “bootstrap” the rest of the network which can operate at higher speeds outside of the Tor cloud. This is similar in nature to the use of public key cryptography, which is notoriously slow. In this case public key cryptography is generally used to negotiate a symmetric key which can the be used for the remainder of the communication.

3 Motivations

Our interest in this subject is motivated by the recent work in bot nets and malware activity. Although Fast Flux networks are most often associated with protecting malicious web sites, our work on the Compute Farm gave us insight on how to apply these techniques to another domain. The Compute Farm topology, with its centralized space and distributed workers maps well to the basic ideas behind Fast Flux networks. Much like the zombie proxy machines in a Fast Flux network, the Compute Workers are highly distributed and can work together to provide an ever changing and shifting network. The Space itself acts much like a DNS server, resolving requests for servers by new clients and acting as a central authority to choreograph the otherwise chaotic network behavior.

By applying the aforementioned techniques to the Compute Farm network, we were able to create a chat protocol that protects the anonymity and security of both the network and the end user. Thus, sensitive communications such as political dissidents or the command and control system of a large bot net could make use of this more secure version of IRC. Ultimately, any text based communication where privacy and availability are a concern, would benefit from this network design.

4 System Features

Our first goal of creating a robust chat network that makes use of disposable components is realized through several features of our system. First, the actual chat network is run entirely upon an unreliable set of Compute servers. Tasks are themselves temporary server instances that are “run” by a Compute server for a designated amount of time, sometimes less. These servers route messages from one to another without the aid of the Space. Clients connect to these servers through proxies that are aware of the flaky nature of these server Tasks. When a particular server goes down a client will transparently reroute traffic to another server without the knowledge of the user. The only permanent component in the network is the Space. This is currently but not necessarily a single point of failure. In addition to this fact, the Space is used very minimally, with its major task being a form of name resolution.

Our second goal is realized both through through the results of the features satisfying the first goal (e.g. a set of short-lived network servers) and the hiding of any permanent components behind the Tor network. These two features make the network nearly impossible to take down. First, by the time a network server is actually located it has served its purpose and can be disposed of. This makes attacking the servers fruitless. Second, finding the location of a permanent component such as the Space, which is a service hidden behind the Tor cloud, is cryptographically hard, thus fulfilling our second goal.

5 Design and Implementation

Our chat network was designed to emulate a traditional Internet Relay Chat network (IRC) on top of the existing Java RMI Compute Farm architecture that was developed over the quarter. We decided to emulate IRC for several reasons. First, the IRC protocol is an established and widely used protocol. This lends itself to adoption and provides a wealth of useful tools that we can reuse and borrow ideas from. Second, IRC is a relatively simple text based protocol that allows for rapid development and parsing.

A high level overview of our design and architecture can be seen in figure 1. This figure demonstrates several things. First, the red arrows show the registration process for server and router tasks. They also show the client proxy’s request for a server in our network. Once several Compute servers have registered and a server handle has been requested by a client proxy, that proxy can then connect to the network. This link is shown in purple in the figure. Finally, a client can connect to the proxy (shown in green) and send a message to a channel or client. This message flow is represented by the blue arrows.

There are several key technical issues that must be solved in order to realize this high-level design. These issues and the resulting architectural decisions that had to be made are discussed in sections 5.1 and 5.2.

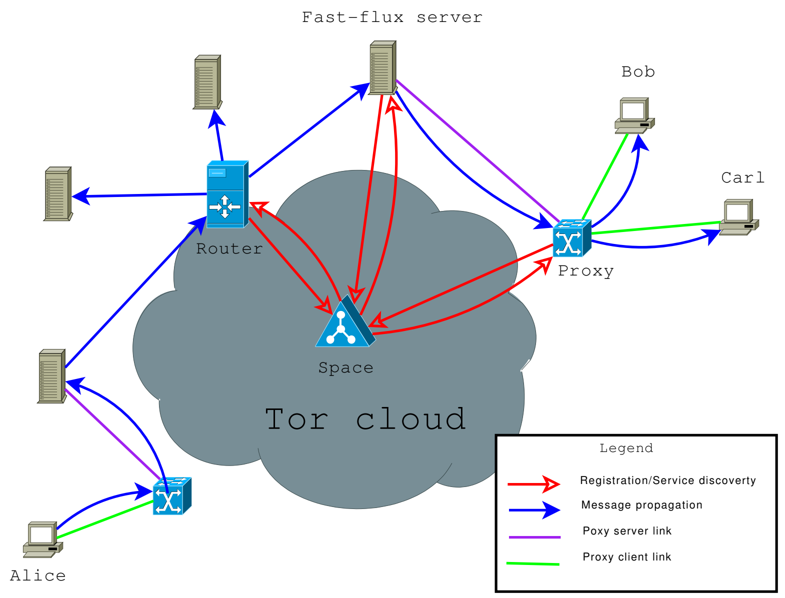

Figure 1: System Overview

5.1 Key Technical Issues

5.1.1 Client View

There is one major technical issue facing a potential client in our network, namely the reconciliation of a client's traditional view of the network as having a stable entry point and the view necessary to maintain a presence on our network. In order to function correctly the client must be aware of, and ready to deal with the flakiness of the network. Otherwise, there must be some buffer between the client and the network that is able to transparently mask this characteristic.

5.1.2 State

One of the major technical issues that the network architecture faces is correctly distributing state among the chat servers. This is also one of they key differences in distinguishing our network with a traditional client and server architecture where state is centralized in a single location. Additionally, the state must be distributed transparently so that the clients maintain the view that they are connected to a single chat server without any knowledge of the distributed underlying network.

5.1.3 Message Routing

Another technical issue that the network architecture must deal with is routing messages between the distributed chat servers. Global state is replicated in each server, allowing for a large degree of independence. This independence is necessary to maintain the fleeting nature of the chat servers, where each individual servers are able to enter and leave the network at anytime. Unfortunately, this constantly changing network architecture must be monitored accurately in order successfully distribute the necessary state to the chat servers. A key technical issue for this network is efficiently distributing this state information in a way that is efficient and not overly complex.

5.2 Key Architectural Issues

5.2.1 Network Proxy

In order to deal with the issues presented in section 5.1.1 we decided to create a client proxy. This allows for two benefits. First, the proxy remains ignorant of the protocol itself making it fairly simple in nature. It operates by querying the Space looking for a server. The Space returns a server handle and the proxy connects to that server. When a client connects to the proxy it cuts the logical connection to the server in half, pushing data from the client to the server and back. When a server dies, the proxy is able to connect to a new server (whose state is recovered from other servers on the network) and continue as if nothing ever happened. Unless there is a major failure in the network, or the Space goes down, the client remains unaware of this underlying network behavior. Second, the fact that we are hiding the underlying network behavior from the client along with the fact that we are emulating IRC allows us to reuse any off the shelf IRC client. Currently this is not the case as we only implement a subset of the protocol. However, once an appropriate subset of the protocol is implemented this would be possible.

5.2.2 Router Tasks

One of the essential components of our network is the router tasks, designed to facilitate message routing between the various chat servers. The router tasks are one of two tasks, the other being the chat servers themselves. The router tasks operate by maintaining a list of handles to all the chat servers that are currently operating within the network. When the chat server wishes to send a message, it forwards it to the router task who will then either send it directly to the destination for unicast packets or broadcast it to all other chat servers in the case of multicast packets. The router tasks function in similar ways to the supernodes in other scalable networks such as the large scale FastTrack network [4].

In the current implementation of the network, the first Computer to connect to the Space is assigned the router task. Any other Computers that connect will be assigned a chat server task. Part of the initialization of the chat server involves requesting the space for a network handle of the Computer that is running the router task. The chat server will then create a persistent connection to the router task which can be used to send and receive messages from any chat server in the network.

This architecture provides efficient network utilization because the chat servers are only required to maintain a single connection to the router in order to communicate to the rest of the network. In addition, the router is the only component that needs to keep track of the fleeting chat servers. The single router in the current implementation is actually one of the single point of failures; however, it could be trivially replicated because it doesn't store any unique state that could not be recreated by the Space.

5.2.3 Consistent State

With the chat servers communicating to each other through the router task, state can be trivially replicated by broadcasting events. For instance, when a client sends a chat message to the chat server it is connected to, that message is broadcasted almost verbatim to the other chat servers through the router. When the other chat servers receive the broadcasted message, they will process it in a manner similar to that of receiving the message directly from a connected client.

Two issues that could destroy the consistency of distributed state is out of order packet delivery and dropped packets. This is partly solved by using the reliability and in order delivery of TCP communication between the chat and router servers. The other part of our solution is that the router task sends out messages in the order they are received. The combination of these two factors guarantees that all chat servers will received messages in the same order that the router sends them which helps maintain consistent state.

Another issue that had to be solved was distributing global state to new chat servers that join the network. This was solved by modifying the router so that it caches all new state that is sent through it. The cached state includes the session id and nick of all connected clients, and the chat rooms they are currently in. When a new chat server joins the network, it connects to the router who in return sends it a copy of the current state of the system. This is done in a single atomic step so no messages that contain state changes will be lost.

6 Conclusions and Future Work

In this report we have presented the design and implementation of a robust chat network that operates under the same principles as a traditional Fast-Flux service network. Our proof of concept implementation shows that it is possible to implement a chat network that satisfies the goals presented in the introduction and demonstrates the usefulness of a Fast-Flux design pattern in situations that demand or would benefit from the anonymity and temporary nature of such a network.

In the future we hope to expand our proof of concept implementation in several ways. First, we want to implement a proper subset of the IRC protocol, or at least emulate it such that we can connect to the network with existing IRC clients. In this way we could drop our network in an existing network and see how it performs in a real world situation. Second, we would like to improve and audit the security characteristics of our system. Some glaring feature holes include end-to-end encryption, lack of login credentials, lack of privilege levels, etc... Third, we would like to improve the message routing protocol and allow for redundant routing servers.

Currently we are using a store and broadcast mechanism which is very inefficient. Finally, we would like to make the Space redundant and implement a P2P style Space cloud. This would take care of our single point of failure and make our network much more robust.

References

[1] `http://www.secureworks.com/research/threats/migmaf/`.

[2] `http://www.torproject.org/`.

[3] `http://www.hacktivismo.com/projects/index.php`.

[4] `http://cvs.berlios.de/cgi-bin/viewcvs.cgi/gift-fasttrack/giFT-FastTrack/PROTOCOL?rev=HEAD&con`

[5] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, 1999.

[6] T. Holz, C. Gorecki, K. Rieck, and F. Freiling. Measuring and detecting fast-flux service networks. In *NDSS*, 2008.

[7] T. H. Project. Know your enemy: Fast-flux service networks an ever changing enemy. Technical report, The Honeynet Project, 2007.