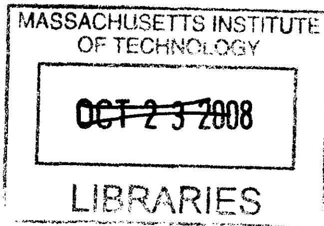
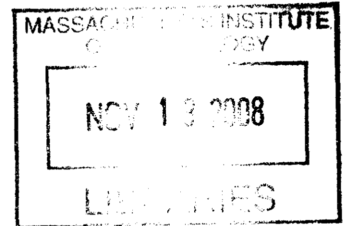
Decentralized Information Flow Control on a Cluster





by

Natan Tsvi Cohen Cliffer

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

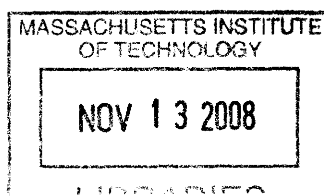
September 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 20, 2007

Certified by ....
Robert Tappan Morris
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students



ARCHIVES

Decentralized Information Flow Control on a Cluster

by

Natan Tsvi Cohen Cliffer

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Information flow control security models can prevent programs from divulging sensitive information in unexpected ways. There has been significant work on tracking information flow between processes in the same computer at the operating system level. I present a modification to the Flume information flow control system for OpenBSD that allows information flow to be tracked between programs on different computers, as long as the system software on all involved computers is maintained by the same trusted entity. This allows the benefits of Flume to be applied to computer systems that take the cluster approach to scaling.

Thesis Supervisor: Robert Tappan Morris
Title: Associate Professor

Acknowledgments

I would like to thank Robert Tappan Morris for his advice on this thesis and the entire project. I would also like to thank Maxwell Krohn and Alex Yip for their constant advice and other help, and Frans Kaashoek, Micah Broadsky, Austin Clements, and Russel Cox for useful input and motivational support.

Contents

1 Introduction **13**

2 Background **15**

2.1 Information Flow Control and DIFC 15

2.2 The Flume Model 17

3 Goals **19**

3.1 Motivation 19

3.2 Assumptions 21

3.3 Challenges 21

4 Abstraction **23**

4.1 Naming 24

4.2 Flow control concerns 26

4.3 Rules 27

4.3.1 Socket creation 27

4.3.2 Connection to a socket 28

4.3.3 Acceptance of a connection on a socket 28

4.3.4 Sending to a socket 29

4.3.5 Receiving from a socket 29

4.4 Example 29

5 Implementation **33**

5.1 Overall Structure 33

5.2 Providing a socket interface 34
5.3 Multiplexing Network Connections 35
5.4 Security and Authentication 38
5.4.1 Trusted network 39
5.4.2 Untrusted network 40
5.5 Limitations 41
5.5.1 Resource exhaustion 41
5.5.2 Communication outside the system 42

6 Evaluation **45**
6.1 Test Setup 45
6.2 Benchmarks 46
6.3 Results and Discussion 47

7 Conclusion **51**

List of Figures

5-1 Anatomy of a Flume connection. The thin arrows represent connected file descriptor pairs; the thick arrows represent messages 36

6-1 Round trip time comparison 47

6-2 Setup time comparison 48

6-3 Throughput comparison 49

List of Tables

4.1 Comparison of point-to-point and rendezvous sockets and their rules . 30

5.1 The network server's RPCs 38

Chapter 1

Introduction

Many computer services are designed to scale by running on a *cluster*, a group of computers administrated by the provider of the service that work together to provide the service with higher availability, the ability to handle more users at once, and/or improved service characteristics for each user[5]. Often, the same services have security policies about which information can be provided to which users.

Decentralized Information Flow Control (DIFC) is a model by which applications can specify which kinds of information can be sent where[10]. Current implementations of DIFC operate at either the programming language level with fine granularity [12], or at the operating system level with process granularity [3] [16], or even as a user space reference monitor, also with process granularity [7]. Flume has the advantage of providing DIFC abstractions in an otherwise nearly standard POSIX environment, so it is an attractive choice for developing flow-controlled versions of existing systems. Flume, however, previously could only provide clustered applications if the parts of the cluster communicated over a shared filesystem, which is indirect and relatively slow, and does not fit the needs of many clustered applications.

This paper describes a set of abstractions in Flume for network communication between confined processes that is inspired by POSIX network sockets, and still preserves the information flow properties of Flume. To achieve both goals, Flume incorporates two different kinds of socket abstractions: *point-to-point* sockets, which behave very similarly to their POSIX cousins once connected but have strong re-

strictions on their naming, and *rendezvous sockets*, which have weaker restrictions on naming, but differ from POSIX behavior in important ways. The combination allows a privileged server process to send information to other processes with a variety of different security policies, without the clients or the server needing to know the other's policy a priori.

The added socket abstractions were implemented as a cooperation between the Flume reference monitor, and an outboard network server. The reference monitor forwards information between connections with checks to ensure ongoing communication is consistent with the flow control rules; the outboard network server matches up TCP connections with the logical Flume socket abstractions they are intended for and keeps track of addressing information.

Enforcing flow control rules with a user space reference monitor imposes some overhead on both the speed of data flow between processes and the total amount of information exchanged. Experimentation shows that the latency and throughput overheads are sufficiently small for many applications, though creating new connections is particularly expensive. The total extra amount of information on the network is constant for every new connection opened; it does not grow with the amount of information sent along the connection.

Chapter 2

Background

2.1 Information Flow Control and DIFC

A major problem in computer security is protecting information from getting into the wrong hands. Typical computer systems interact with more than one category of information, each category requiring specific rules on which users or programs can access it (examples [2] [8]).

One of the methods proposed to help protect information security is *mandatory access control*, where the environment in which a program runs limits the information the program can divulge to other programs to a set of information the program is deemed to be allowed to divulge. These restrictions can be enforced as a part of a programming language such as Jif [12], by the operating system as in SELinux[14], or by a user-level reference monitor working closely with the operating system, as in Flume[7].

The models provided to describe security limitations also vary. Centralized mandatory access control systems like SELinux require a system administrator to explicitly set the categories of information each program is allowed to access[14]. Centralized access control like this is useful for containing established systems where the information flows and principals are well known a priori.

This paper focuses on extending Flume, which uses a Decentralized Information Flow Control (DIFC) model. In a DIFC model, the allowed information flows are

defined by the applications themselves[10]. Each application can create *tags* that represent categories of information, and define rules for how information with a tag it owns may propagate[10].

A DIFC security model implementations include the security-typed programming language Jif [12], custom operating systems such as Asbestos and HiStar [3] [16], and the mostly user-level Flume[7].

In Jif, all computation is expressed in an extension to Java where the type system is extended so programs can be statically checked for confirmation to the security model. This has the advantage of providing provable information hygiene behavior of the system at a fine-grained level, within the abstraction of the Jif virtual machine [12]. Jif also has the advantage of a system for distributed computing that preserves information flow control[15]. However, a lot of software that could benefit from a DIFC model is implemented in other languages, and might not be suitable for porting to Jif.

Asbestos and HiStar allow arbitrary programs to execute, subject to operating-system level restrictions on communication with other processes. They provide coarser-grained information flow controls, monitoring the flow of information between processes in the operating system [3] [16]. Flume is a similar system, but runs in user space under OpenBSD, allowing stock OpenBSD code to be run with a minimum of modification, as long as it links against the Flume libraries [7]. In these DIFC implementations, systems typically consist of a set of more-trusted processes that set up the flow control policy, and a set of less-trusted workers which do the body of the computation and forward results for declassification by the more-trusted processes [3] [16]. The goal in splitting the work this way is the principle of least privilege; each part of the system should be allowed to do no more than is required to accomplish its task [13].

2.2 The Flume Model

The Flume DIFC model describes the allowable flow of information between *processes* through *endpoints*, which each have a tuple of *labels* describing the allowable flows. Labels, in turn, are made up of *tags*, which describe individual categories of information.

A Flume *process* is more general that the standard use of the term in computer science; it includes both operating system processes and any other abstraction that can store and divulge information. Specifically, files in Flume are modeled in the same way as processes. Writing to a file is modeled as "sending" to the file, and reading from the file is modeled as "receiving" from it.

Any way Flume process can interact with another is marked by an *endpoint.* Endpoints may be readable, writable, or both.

Each endpoint and each process have *labels*, which are sets of *tags*, are opaque identifiers allocated by the system at the request of any process. Tags are generally used to represent types or properties of information—for example, "Alice's financial information" or "written by Bob". Each process and or endpoint p has a tuple of two types of *label*: a *secrecy label*, $S(p)$, and an *integrity label*, $I(p)$. Processes use tags to represent categories of secrets or categories of integrity, depending on what labels the process puts the tags in.

Additionally, each process p has an *ownership label* $O(p)$, which contains the set of label change capabilities belonging to the process. For each tag $t \in T$ there is a corresponding *add capability* $c^{+}(t)$, which allows the process holding the capability to add the tag to its labels, and a *remove capability* $c^{-}(t)$, which allows the process holding the capability to remove the tag from its labels. When a process creates a new tag, it receives both of these capabilities with respect to that tag, and may also optionally grant either or both of these capabilities to all other processes (the tag can become *default-addable* or *default-removable.* Any process may also send any capability it owns to another process (along a legal communication channel), to be placed in the second process's ownership label. The set of add capabilities can be

referred to as $O^+(p)$, and the set of remove capabilities as $O^-(p)$.

For a process p to send information on a writable endpoint e, e must have in $S(e)$ at least all the secrets p has and cannot remove from $S(p)$, and p must have or be able to add to $I(p)$ all categories of integrity e has in $I(e)$. In other words, the condition for p to safely send information on e is:

$$S(p) - S(e) \subseteq O^-(p) \wedge I(e) - I(p) \subseteq O^+(p)$$

Correspondingly, the condition for p to receive data from e is:

$$S(e) - S(p) \subseteq O^+(p) \wedge I(p) - I(e) \subseteq O^-(p)$$

For information to flow between processes, the rule is that it must be able to flow in the correct direction between each process and the corresponding endpoint, and between the endpoints. The rule for information flowing between endpoints is similar to the rules above: if information is to flow from endpoint e to f, e must have a subset of f's secrets, and a superset of f's integrity:

$$S(e) \subseteq S(f) \wedge I(f) \subseteq I(e)$$

As proven in [7], this enforces that all information flows such that no secrets are divulged to processes that have not added all the secrecy labels their source has and cannot declassify, and that no information comes to an integritous process without coming from a source that is able to vouch for that level of integrity.

For further information about the Flume view of DIFC, see [7]. In the rest of this paper, *label tuple* will refer to the pair of S and I labels associated with a particular endpoint or process.

Chapter 3

Goals

3.1 Motivation

Many of the projected applications of DIFC are networked services—computer systems that should be constantly available to multiple users. Flume uses FlumeWiki as an application example[7], and Asbestos uses the OKWS web server as its example[3]. Indeed, the concept of information flow control naturally lends itself to situations with multiple users interacting with a shared computing resource. Networked services often require the ability to scale by orders of magnitude, as a dedicated user base slowly grows[4], or as a flash crowd temporarily causes a spike in traffic[1].

One of the most effective ways of dealing with such scaling requirements is distributing the responsibility for the service among several PC-class computers in a cluster[5]. Under the architecture described in [5], software components are divided into several classes (in a somewhat simplified model):

Front ends which accept requests from the outside world and forward back responses.

Workers which accept jobs from the front ends, and do the bulk of the work.

Customization Database which stores persistent information the workers may need to query.

Load Balancer (in [5], the *Manager*, which allocates work among the instances of the software components.

Each of these components can be separated off onto different processors, communicating over a network.

Though Flume currently supports some kinds of clustering by providing the ability to share file servers, none of the current operating system-based DIFC systems allow a safe and information-tracked way to scale a service with communication requirements that can not be fully served by a shared file system. Without information tracking over network connections, the worker components in the example system design would have to be trusted as declassifiers for all secrets they own, as they would have to talk to the database server over the network. Alternately, the database server would have to be on the same physical machine as all of the workers, which would limit the amount the service could scale. Similarly, either workers and front ends would have to be on the same machine, or the workers would have to be able to declassify their information themselves, instead of following the more secure plan of allowing the front ends to declassify the information.

Flume cluster sockets solve this problem by allowing the reference monitors on the source and destination computers to cooperate to enforce Flume information flow constraints between processes. The database server can label outgoing data just as it would for local communication, and the two reference monitors will ensure that only processes with the appropriate labels will see the information. Similarly, the front ends can be in charge of declassifying the information, allowing the worker processes that feed information to them to run without the requirement for declassification-level trust.

More generally, Flume is used for complex systems composed of many interacting processes sending messages to each other. When such a system needs to scale, the processes are often distributed among several different computers. This suggests that for Flume systems to scale transparently, Flume needs to support controlling information flows over network connections.

3.2 Assumptions

There are limitations on the security model any networking extension to Flume can provide. It is currently not possible to conclusively verify whether or not an unknown computer across the network is faithfully obeying all of Flume's constraints. This means that any information-tracked network extension to Flume must only allow information-tracked exchanges of information between computers that are *trusted* to be validly running Flume, and to have no other compromising software loaded. Exactly how this trust is established is beyond the scope of this document, but I will assume that the administrator of all the computers can be treated as a single entity, who can be responsible for out-of-band key propagation so the reference monitors can authenticate with each other.

3.3 Challenges

The main challenge the Flume networking extension faces is *covert channels*, ways processes can communicate with each other using aspects of the system that are not normally used for communication, but that are difficult for the system to control [11]. The most important goal of an information-tracked networking addition is the elimination of covert channels between any pair of processes confined by the Flume system. One challenge involved in this process is the naming of these connection points. A flat naming scheme like the port number of a POSIX Internet domain socket does not suffice; taking up a name in this space is tantamount to communicating that name to all other processes using the namespace. This is because a successful name reservation attempt informs the process making the attempt that all other processes able to make that attempt have not yet done so, if the attempt succeeds. It also informs all other processes that some process has now made the request to reserve that name, because any future legal request to reserve that name will now behave differently. This is a bidirectional channel, since attempting to take a name is a method both of sending and receiving information, at a rate of one bit per name

(whether the name is taken or not).

In general, if a process can reserve a name for itself in such a way that subsequent attempts by other processes to reserve the same name will behave differently, the process reserving the name must have the same secrecy and integrity labels as any other process that can attempt to reserve the name. Since this namespace communication is bidirectional, any namespace must be partitioned according to the complete label tuple for the privilege of reserving names.

This limitation also suggests another problem: if the namespace is completely partitioned along label tuple lines, two processes that wish to communicate with each other must first know the exact set of labels at which to rendezvous. Communicating this information requires a channel that does not require taking up a name in a namespace.

Chapter 4 describes the way the networking extension to Flume instantiates such a partitioned namespace, as well as a solution to the problem of two processes safely rendezvousing at a name in the absence of each of them having prior full information about the label tuple of the other.

Unfortunately, Flume sockets do not currently deal with all sources of covert channels. As with any computer system, bandwidth is a limited resource, and Flume shares it among all processes, which presents an open covert channel. Also, timing attacks could be used for a confined process to convey information to an unconfined process that can listen to the network. These limitations are discussed further in chapter 5.5.

Chapter 4

Abstraction

Flume sockets, for secure communication between Flume processes on different computers, were designed with two major goals in mind. First of all, the abstraction should be easy to use, and as consistent as possible with standard POSIX utilities and other Flumeprimitives. Second, the abstraction should minimize the chances of a process forming a covert channel. To this end, two complimentary kinds of sockets are available, each using slightly different semantics. *Point-to-point* (p-to-p) Flume sockets allow information-tracked one-to-one communication streams between processes. However, restrictions in the creation of point-to-point Flume sockets make rendezvous between a client process (one that connects to an established server) and a server process (one that listens for connections to a given socket name) difficult, so another form of socket is available: Flume *rendezvous* sockets. Rendezvous sockets are one-way and may duplicate information to multiple receivers, but allow a form of one-way rendezvous that would be difficult to reconcile with Flume information-tracking rules otherwise. A summary and comparison of the rules for p-to-p and rendezvous sockets is found in figure 4.3.5.

The general model of (successful) Flume socket connection is that a confined *server process* creates a *listening server socket*, which is represented by a unique file descriptor and intrinsically bound to a name n. The server process must also call `listen` on this socket, to specify the number of simultaneous requests to queue. Then a *client process*, possibly on a different host, may *connect* to the server socket by

giving a name c that the server socket *answers to.* For p-to-p sockets, c must be equal to n for c to answer to n. For rendezvous sockets the rules are more complicated, and covered in section 4.1. The server process can now *accept* the connection, which creates a *connected server socket*, represented by its own unique file descriptor and associated with n. On the client process side, the connection operation produces a *connected client socket*, represented by a different unique file descriptor and associated with c. The two processes can now communicate with each other by writing to and reading from their respective file descriptors, though some such reads and writes may not be allowed by the DIFC rules (see 4.3.5).

4.1 Naming

Namespaces are one tricky area with respect to covert channels in information flow control systems. To deal with the communication implicit in reserving a part of a namespace, names must be allocated randomly, allocation of names must be limited by the process's labels, or multiple objects must be able to be meaningfully named the same thing. The second option corresponds to p-to-p Flume sockets, and the third option corresponds to Flume rendezvous sockets.

Point-to-point Flume sockets are named by a triple of elements: the name of the computer the server is on, a port number to differentiate between similar sockets, and a label tuple associated with the socket. The label tuple is part of the name of the socket, and also governs access to the socket. A process can only create a p-to-p Flume server socket with a label tuple it can have bidirectional communication with, thus side-stepping the chance for a naming covert channel.

Incorporating the label tuple in the name of a resource is in fact a general way to avoid this problem if you are willing to accept the creation restriction on the names, and is used in other forms in other parts of Flume: the integrity namespace for the file system is another instance of this pattern, albeit in a somewhat more complicated form.

Unfortunately, this restriction on naming creates problems for a server of many

clients, each with a different set of secrets. Using only p-to-p Flume sockets, the server would have to anticipate the label tuple of each requested connection to arrange a socket to be listening there, or each client would have to know the tags of the entire set of secrets the server has the right to declassify, to put all those tags in the name of the socket to connect to. The client, having just a subset of these labels, would then be able to send the server an address of a less-tainted socket on which to connect. Neither of these plans are elegant, so a slightly different abstraction is useful for arranging rendezvous at a p-to-p Flume socket that does not require the database client process to know the exact set of secrets the server knows.

Because only one p-to-p listening server socket can exist with a given name at a time and connections to p-to-p sockets refer to specific complete names, any connection attempt unambiguously refers to only the one listening server socket. The connection formed will have only two endpoints involved: the connected client socket, and the one connected server socket. This connection generally behaves like a connected POSIX stream socket, but is also subject to DIFC rules that may restrict one or both directions of information flow at any given time.

Flume rendezvous sockets are also created with a specific host, port, and label tuple triple, but any number of Flume rendezvous sockets can be created with the same triple, which is the third way listed above of avoiding creating a namespace-based covert channel.

A connection to a rendezvous listening server socket allows all sockets on the specified host and port that are allowed to receive the information accept the connection. The resulting connection is one way—the client may send to the servers, but the servers may not send to the clients. It is also multicast—every connected server socket associated with the same connected client socket gets its own copy of the information. In this way, Flume rendezvous sockets somewhat resemble a broadcast datagram model. They allow a confined server process to bind to a rendezvous socket with all its secrets, yet still receive from confined client processes sending with respect to their specific secrecy labels. If two-way communication is needed between server and client, the clients can send to the rendezvous socket the name of the p-to-p Flume

socket where the client would like the server to listen, and then reconnect to the new socket. This is the expected use of rendezvous sockets, from which they derive their name.

Rendezvous sockets differ from a datagram model in that they do guarantee order of received messages.

4.2 Flow control concerns

To a first approximation, p-to-p Flume sockets implement the Flume DIFC rules as endpoints with the labels given in their names. The DIFC rules govern which ways information can be sent, based on the processes' labels at any given time. However, there are some details that require further definition from this rule.

When a process successfully connects to a socket on which a server is listening, the server's side becomes readable, and an `accept()` call will not block. This transmits the fact that some process tried to connect to the server. Therefore, even if the client only intends to read from a socket once connected, it still may not connect to a socket it does not have the ability to write. For the same reason, a server may not bind a socket to an address it does not have the ability to receive data from. This combines with the namespace restriction to require a process to be able to both send to and receive from a socket to bind to it. These constraints apply to both p-to-p and rendezvous Flume sockets.

Another possible source of a covert channel is the status of a connect call. With standard POSIX sockets, an attempt to connect will fail if there is no such socket bound at the other end. However, Flume sockets can only afford this luxury some of the time—the return code of the POSIX-style `connect()` request transmits information from the server to the process that called `connect()` (the client), which may have a label such that it can only send to the socket. specifically, it transmits the information of whether or not the server has created the socket in question. Therefore, requests to connect to a p-to-p Flume socket must only return a value that depends on the server's status if the client is allowed to receive information from the

socket. If the client is only allowed to send such information, the connection request will only fail if the physical connection between the two hosts is down. If there is no server listening, any information the client sends is discarded, and the client is given no indication of this fact. All Flume rendezvous sockets are one-way, and thus all connection attempts succeed according to the clients, and the resulting sockets are always writable.

When a one-way connection is formed, the servers at the other end might accept a connection but be late in reading data. In the absence of a two-way connection, Flume stream sockets do not guarantee delivery, as doing so would essentially require an infinite queue. Instead, once the queue fills all further information sent is discarded until the server processes start consuming information.

4.3 Rules

To fulfill all of the above constraints, the Flume sockets follow rules for all their operations. In the following formal rules, a process p *can send to* a socket named with s if $S(p) - S(s) \subseteq O^-(p) \wedge I(s) - I(p) \subseteq O^+(p)$, *can receive from* s if $S(s) - S(p) \subseteq O^+(p) \wedge I(p) - I(s) \subseteq O^-(p)$, and *can have bidirectional communication with* s if $S(p)-S(s) \subseteq O^-(p) \wedge I(s)-I(p) \subseteq O^+(p) \wedge S(s)-S(p) \subseteq O^+(p) \wedge I(p)-I(s) \subseteq O^-(p)$. In all cases s refers to the socket name, which includes a label tuple. This label tuple is guaranteed to be the same as any endpoint that refers to that socket.

4.3.1 Socket creation

For Flume sockets, *creating* the socket is equivalent to the combination of a POSIX-style call to `socket` and `bind`. For simplicity, the RPCs provided to the RM combine these two operations for the purposes of creating server sockets. `socket` and `bind` can still be modeled as separate in the library a confined process uses to access Flume functionality—the current Python Flume library separates them this way.

A process p may create a p-to-p listening server socket named with s if p can have bidirectional communication with s, and no socket already exists with the same

address, port, and label tuple as s. A failure by p to create the socket s due to the s already existing is only indicated as an existence error if p has permission to create s.

A process p may create a rendezvous listening server socket named with s if p can receive from s. This creation may succeed regardless of how many identically-named sockets are in play.

If p is not allowed to create the socket, then the creation attempt will indicate a permission error.

4.3.2 Connection to a socket

Like the case for creating sockets, the Flume RPC for *connect* simultaneously allocates a socket and attempts to connect it. The functionality of `socket` and `connect` may again be separated in the library, but for this section *connect* refers to the Flume RPC.

A process p may connect to a socket named with s any time p can send to s. For p-to-p sockets, s will successfully connect only to sockets that match all elements of the name s. For rendezvous sockets, s will connect to any sockets n that match the host and port of s such that n's label can receive from s's label.

If s is a p-to-p socket and p may have bidirectional communication with s, p will be returned a value that reflects whether s exists at the remote host and there is a process listening. Otherwise, p will not be returned a value containing this information; the connection attempt will appear to succeed, and the RM will discard all information written to the resultant "fake" socket end.

4.3.3 Acceptance of a connection on a socket

A process p may accept a connection on a socket named with s at any time p can receive from s. The new socket's label tuple is identical to that of s.

4.3.4 Sending to a socket

A process p may send to a p-to-p socket named with s any time p can send to s. p may send to a rendezvous socket named with s if p can send to s and p is the process that connected to s, not the process that created it. If the send "succeeds" but the process on the other side of the socket cannot receive the information according to 4.3.5, the information is silently discarded by the RM on the foreign host. The call to write the socket fails with a permission error if the process is not allowed to write to the socket, and returns the number of bytes written otherwise.

4.3.5 Receiving from a socket

A process p may receive from a p-to-p socket named with s any time p can receive from s. p may receive from a rendezvous socket named with s if p can receive from s and p is the process that created s, not the process that connected to it.

4.4 Example

As an example of how both p-to-p and rendezvous Flume sockets can be used, consider a web service set up with multiple computers acting as HTTP server front ends, and one database server supplying them with dynamic information, some of which contains secrets belonging to particular people signed in to the web front ends at the time. As in previously described Flume systems [7], the web servers would be designed to spawn alternate processes to deal with particular logged-in users, and the alternate processes would limit themselves to the tags associated with those users except at the very last stage where the raw HTML response will be sent to a declassifying process to be sent back to the web browser.

So far, this scenario is very similar to the FlumeWiki example used in [7]. However, when the information that the server needs to query is not stored in flat files on a shared disk, some way of allowing the web servers to query the shared store of information is required. In this scenario, there is a single database server trusted

	Point-to-Point Sockets	**Rendezvous Sockets**
Client Side	one	one
Server Side	one	many
Directionality	Bidirectional, labels permitting	Always client $\rightarrow$ server.
Namespace	Each socket takes a unique (host, port, label tuple) triple in the namespace. Allocation is only allowed to a process that can send and receive to the label tuple	Any number of sockets can be bound to the same (host, port, label tuple) triple. Each socket gets all incoming information duplicated to it.
A listening server socket named with n answers to a connection to the name c	If $c = n$	If c and n have the same host and port, and c's label tuple can send to n's.
Process p can bind to socket with label tuple s	p can have bidirectional communication with s	p can receive from s
Process p can connect to socket with label s	p can send to s	p can send to s
`connect` to a socket with label s returns p a meaningful value	p can have bidirectional communication with s	never
Process p can accept a connection on a socket with label s	p can receive from s	p can receive from s
Process p can send to socket with label s	p can send to s	p can send to s and p connected to s
Process p can receive from socket with label s	p can receive from s	p can receive from s and p created and listened on s

Table 4.1: Comparison of point-to-point and rendezvous sockets and their rules

with declassification capabilities for all secrets, which feeds information to the various worker processes on other computers. A worker process may be trusted with declassification capabilities for the specific secrets with which it deals, or it may be trusted with no declassification capabilities, and instead communicate with a sepa-

rate front-end process that does have the capability to declassify the few secrets it will see. This structure greatly decreases the amount of code that could improperly leak information to only the database server and front end, making it easier to be confident the application as a whole has no data leaks.

The database server p_d initially binds a single rendezvous socket to port r, with the secrecy label S_S consisting of all the secrets in its database. Since any client process of the database server is expected to have a secrecy label $C_S subset S_S$ consisting of a subset of this, any client will be able to connect to this socket and send the label C at which it would like to meet for two-way communication (information can flow from less-secrecy to more-secrecy contexts). The database server then binds a p-to-p Flume socket to this label at port y with the labels C as requested by the client, and the client connects to this new port. The socket at $y : C$ is now two-way, because the database server has the capability to remove all but the correct tags from its secrecy label.

Because this two-step rendezvous procedure (and Flume socket connection in general) is somewhat expensive, the front end should keep a persistent connection to the database server open for all its requests, to save on overhead.

Chapter 5

Implementation

5.1 Overall Structure

Flume is implemented as a user-space reference monitor which proxies programs' requests to the operating system. Confined processes run under systrace, which limits their system calls to "safe" operations, as well as requests to the reference monitor. The reference monitor keeps track of processes' labels, and executes requests in a safe manner based on those labels. Additionally, the reference monitor stores the label information that must be persistent across reference monitor restarts with an *identity database* or IDD. Confined processes are linked against a special version of libc that replaces the system calls for the unsafe standard operating system functions with safe calls to the reference monitor.

The logical reference monitor itself consists of several processes: the main reference monitor process itself (which we will from now on refer to as the RM), and various servers that handle specific modular aspects of the reference monitor's interaction with the outside world. For example, each file system that a confined process can access is represented to the RM as one file server process. Each of these servers, both subsidiary and main RM, is implemented as an asynchronous RPC server using the Libasync/Tame software library in C and C++.

The RM itself handles requests for information about labels, proxying direct connections between processes in an information-tracked way, and delegating appropri-

ate requests to the secondary servers. See [7] for more information about the overall structure of Flume.

The networking extension to Flume consists of two main parts: a generalization of the file descriptor proxying code and adapters to provide an interface for network sockets in the main RM, and a specialized server for multiplexing network connections. The networking extension also contains additions to the Flume version of libc to interface with the reference monitor, and some setup of optional ssh tunnels to provide security against adversaries able to access the local cluster network directly.

5.2 Providing a socket interface

When a confined process wishes to form an information-tracked network connection, it does so by a series of RPCs to the Flume reference monitor, rather than directly making system calls (which are prohibited to confined processes). We shall investigate each side of a connection independently, the server's listening socket, and the client's outgoing connection request.

To create a socket for listening, a process specifies the address on which it wishes to listen in a socket creation request to the RM. If the address is allowable according the the rules established in the Flume socket semantics defined in section 4.2, the reference monitor creates a socket pair with which to proxy the connection, and forwards the request to register a socket on to the network server. If there are no problems at the network server end, the RM responds to the RPC to create a socket with one of the ends of the socket pair it created, reserving the other end in a structure representing the socket end to use later.

The reference monitor includes several types of socket end structures. The socket ends are divided as to whether they are meant for communication or listening for sub-connections, and by method of connection. This paper deals with both listening and communication socket ends, but is specific to the socket ends intended for communication over the network. Each socket end has a unique opaque handle, which a confined process can use to refer to the socket end in a way that does not convey

information to the confined process.

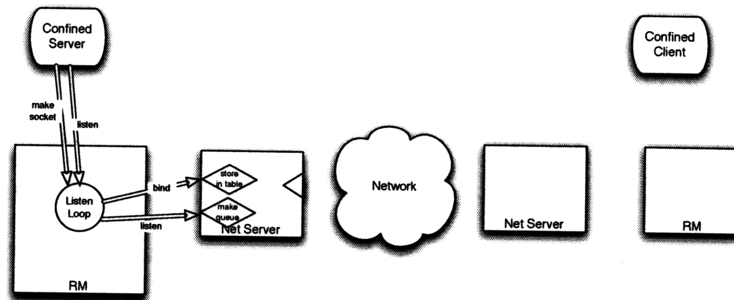
When the server process requests to listen on the newly created server socket, the RM looks up the appropriate socket end in a table of sockets, and starts an asynchronous listening loop on the socket end. The listen loop continually makes RPCs to the network server for the next incoming connection on that given socket. For each new connection, the listen loop creates a new socket pair with which to talk to the client, then starts up a proxy loop between one end of this new socket pair and the incoming connection file descriptor. The proxy loop essentially repeats information by reading from each side and writing to the other, but filters the information sent based on the current status of the process's label as compared to the socket's label.

To allow the confined process to retrieve the proxied file descriptor for the new connection, the RM sends the opaque handle corresponding to the socket end to the process over the listening socket. Over on the process's side, the libc has been modified so that an `accept()` on a Flume socket reads the opaque handle from the socket, then sends an RPC to the RM to retrieve the corresponding file handle.

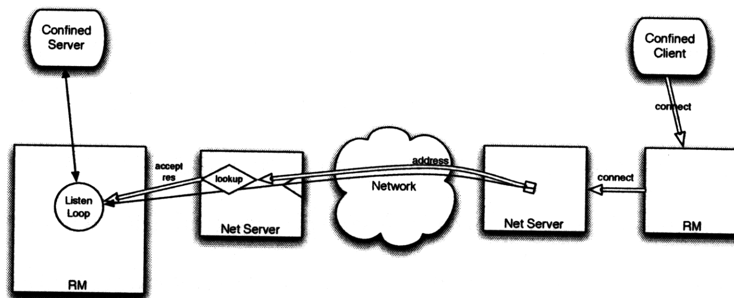
To connect to a socket as a client, the process also specifies the address it would like to connect to in an RPC to the RM. The RM then ensures this connection attempt is legal, and makes its own connection RPC to the network server. If the connection RPC is successful, it returns a file descriptor for the connection to the foreign server. The RM then creates a new socket pair, connects one end to the new outgoing file descriptor in a proxy loop, and sends the other end of the new socket pair back to the client process in the response to the RPC.

5.3 Multiplexing Network Connections

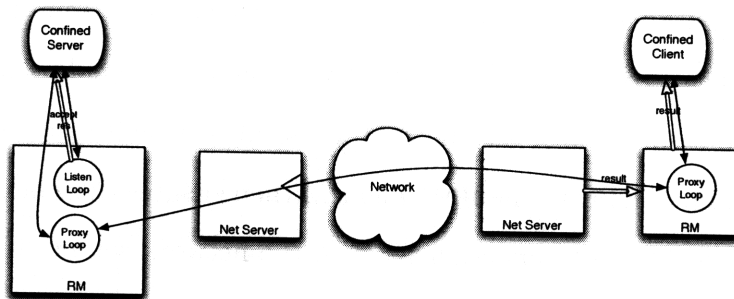
The Flume network server is a distinct process that is in charge of multiplexing listening network connections for the reference monitor. It maintains and listens on a network port for incoming connections from any other Flume computer, and redirects the incoming connection to the appropriate location based on the Flume address. It also acts as an RPC server to the RM, allowing the RM to specify which logical Flume



(a) The server binds and listens to a socket



(b) The client connects to the socket



(c) The server accepts the connection

Figure 5-1: Anatomy of a Flume connection. The thin arrows represent connected file descriptor pairs; the thick arrows represent messages

sockets are bound to which Flume socket addresses, and allowing the RM to request socket connections on behalf of confined processes.

The network server keeps a bidirectional mapping of Flume address of computers against their IP-level network addresses. Client programs only need specify the Flume domain address of a computer to which they wish to connect. Additionally, the network server keeps a mapping of server socket handle to the address on which the socket is listening, and separate mappings for the converse: one set of mappings of

specific address to point-to-point listening Flume socket end handle, and another set of mappings from only the port number to a set of handles that represent all Flume rendezvous sockets listening on that port at all. Finally, the network server keeps a queue structure for every listening socket, in which is stored file descriptors for incoming connections.

The network server accepts five kinds of RPC from the RM. In the lifecycle of a connection, the RM first request the network server will receive will be a `bind`. This request takes in the opaque handle the RM has decided corresponds to a particular socket end, and associates it in the network server with a given address, whether p-to-p or rendezvous. Second, the `RM` sends the network server a request to `listen`, at which the network server sets up the incoming connection queue for that particular socket. The `bind` and `listen` calls could be combined into one call, but POSIX tradition specifies that the socket namespace is taken up (`bind`) at a different time than the length of the listen queue is specified (`listen`). The RM will automatically at this point make a `accept` request to the network server, even though the confined process has not specified an `accept RPC`. This accept call will return to the RM when an incoming connection is formed.

Finally, the RM may send a `connect` RPC to the network server, specifying that the network server should make a connection attempt to a foreign server, and send along the addressing information for the Flume socket required at the other end as the first thing on the new socket. If the connection attempt does not succeed, or sending the addressing information fails for some reason, the network server signals a connection error to the RM, which the RM will report back to the confined process. Alternately, if the connection attempt succeeds, but the foreign network server cannot find the addressed socket, the local network server replies to the RM that the socket was not found. The RM may or may not inform the confined process of this, based on the rules in section 4.2.

The RPCs the network server accepts are summarized in table 5.3.

When a request for a new connection comes in over the outward listening link, the network server first executes a challenge-response authentication (see section 5.4),

`bind`	Sets up a mapping between a socket end number and an address
`listen`	Sets up a connection queue for a given socket end number
`accept`	Call only once `listen` is called on an open socket; returns the next connection attempt
`connect`	Requests a connection with a socket bound to a foreign server
`close`	Tears down the structures corresponding to a particular server socket end.

Table 5.1: The network server's RPCs

then reads and decodes the addressing information that is the next thing over the new file descriptor, and looks up the socket or set of sockets for that address. If the address is for a p-to-p Flume socket, the network server enqueues the file descriptor in that socket's queue. If the address is for a rendezvous socket, the network server first constructs a new socket pair for each listener on that port that can read the incoming connection's label tuple, and connects one end of each of these new socket pairs to the incoming socket with a repeater loop. This allows each listening process to receive information from the foreign client independently of all the others. The network server enqueues the receiving end of each of these new socket pairs in one of the listening file descriptors' queues.

5.4 Security and Authentication

To ensure that information between confined processes on different computers is always properly tracked, the Flume system must ensure that the following security criteria are fulfilled:

- No process other than another network server approved by the cluster's administrator may connect to a network server as a peer.
- All connections between Flume confined processes with any secrecy or integrity are set up through the network servers.

- Once a connection is set up through the network servers, the flow of the connection remains between the file descriptors through which the connection was initially set up (no TCP hijacking type attacks occur [9]).
- Nobody except the designated endpoints of a connection is allowed to know what information is passed along that connection.

The methods for this depend on the security model for the network on which the Flume cluster is running. Depending on whether or not `ssh` tunnels are being used, Flume sockets are designed for two different threat models.

5.4.1 Trusted network

On a *trusted network*, all the computers with unlimited access to the network the Flume cluster sits on are administrated by the same principal. All processes with root access to any of these computers are trusted not to compromise the above criteria, and all non-root processes are restricted from having raw access to the network below the transport layer. Within the local network, each computer is assigned an address, and all messages are faithfully delivered to the computer to which they are addressed. The local network itself may be connected to other outside networks, but the connection method must ensure that no communication internal to the local network leaks to the outside world, and must firewall off the ports used by the Flume network servers from the outside world.

Given a trusted network, relatively simple security measures suffice. In the simple case, the only measure that needs to be taken is an authentication step to ensure that a connection attempt is made by a proper Flume network server, rather than a Flume-confined process that has clean enough labels to have TCP access to the network.

To this end, as soon as the outward-listening socket in a network server gets a connection, it will send a random challenge to the connecting network server. The connecting network server concatenates the challenge to a shared secret, and sends a cryptographic hash of the concatenation back to the challenger. The challenger can

then compare the response to its own calculation of the hash of the concatenation of the challenge and the shared secret, and allow the connection only if they are the same. The distribution of the shared key is performed out-of-band by the system administrator. When a network server connects out to another, the reliability of addressing on the local network ensures that the process it is connecting to is a valid network server. On the other side, the authentication step assures the listening network server that the incoming connector is valid.

System call interposition ensures that confined processes cannot access the network except through the network servers. The fact that all processes that can access the network in a low-level way are trusted in the trusted network scenario ensures that the Flume connections act as exclusive streams without any snooping; there can be no TCP hijacking or snooping without low-level network access.

A simpler and lower-overhead scheme might involve sending the shared secret over the connection, but such a scheme is more sensitive to misconfiguration errors. If a good network server improperly tried to connect to an adversarial network server just once in this case, the adversarial server could pretend to be good, without detection, and the shared key would have to be changed across the whole system. With the a challenge-response scheme, the bad server could pretend to be good temporarily, but the good server could be reconfigured with an accurate server list without revoking the system-wide key.

5.4.2 Untrusted network

Flume sockets can also be configured for a wider network, one in which there may be arbitrary untrusted computers along the path between any two Flume servers. The untrusted computers can insert whatever packets they like on the network. The Flume servers themselves still must be as trusted as they are in the trusted network scenario.

In the untrusted network case, the network servers can set up a logical secure network by establishing encrypted `ssh` tunnels between all the servers at startup time. The administrator arranges `ssh` keys so that only valid Flume servers can

make this kind of ssh connection to each other, and sets up local firewalls on each computer so no other computers can connect to the port on which the network server is listening.

The `ssh` arrangement creates the same guarantees the trusted network does: the participating computers are all addressed accurately, and the `ssh` connection is encrypted so it cannot be listened in on or hijacked. Given these properties, the same authentication routine ensures that all the desired criteria are met.

This arrangement requires an $O(n^2)$ connection cost in the number of nodes, however, so relatively large clusters of Flume nodes are advised to establish a secure local network. Future versions of the Flume networking extension could establish these tunnels on-demand, with provisions to stay open for some fixed time after they are no longer needed, then close. This would occasionally require a longer setup time for a Flume connection, but would allow the system to amortize the cost of protecting the network across all connection attempts, rather than paying the maximum such cost up-front at startup time.

5.5 Limitations

Unfortunately, there are still some ways confined processes can covertly communicate with each other against the flow control rules using Flume sockets, or even communicate information outside the system against the rules. These channels are difficult to use, however, and may or may not be a concern for specific deployments of Flume. This section describes limitations in the design; performance issues will be evaluated in section 6.

5.5.1 Resource exhaustion

In distributed information flow control, any shared limited resource is a possible source of covert channels[11]. Unfortunately, bandwidth is a shared limited resource. One pair of processes could modulate the amount of bandwidth they take up. If the bandwidth varied from negligible amounts to a large proportion of the available

bandwidth of the system, a second pair of processes could use information about the rate at which they can send to each other to deduce something about how the first pair of processes is modulating its bandwidth use. This would convey information from the first pair of processes to the second, regardless of information flow constraints.

Techniques to ameliorate this resource exhaustion channel typically include specifying quotas on the use of the resource—in this case, rate limits on transmissions. With an unlimited number of processes competing for a limited resource, however, such measures are imperfect. They either allow the number of shares of the resource to hit a maximum, or they start allocating detectably imperfect shares of the resource if too many processes request shares. Since any limitation on the number of processes would count as another limited shared resource, a complete solution to this problem is unlikely. However, for the specific case of network bandwidth, the problem may be less critical than it sounds, because available network bandwidth is prone to large random (from the standpoint of processes) variation in the total amount available, which would add noise to any attempt at a network bandwidth-based covert channel.

Due to the absence of any controls on other resource exhaustion channels in Flume, rate limiting Flume network connections is a relatively low priority. It is not currently implemented.

5.5.2 Communication outside the system

Unfortunately, covert channels of the second kind, to listeners outside the system, can be extremely difficult and inefficient to avoid, especially against an adversary with access to intermediate network points between the two Flume computers. Specifically, a listener could pick up information from a pair of confined processes with secrets communicating with each other by measuring the amount of traffic between the two hosts. Furthermore, if a process with some secret information is actively trying to transmit this information to an adversary that is unconstrained by Flume and can sniff the intervening network, the rogue Flume process does not need to succeed in connecting to a foreign socket, it can just modulate the timing of its attempts to do so.

Some methods for network anonymity are applicable to decreasing the bandwidth of these channels or eliminating them. For example, the Flume system could delay some incoming packets and insert *chaff* (meaningless packets) to other servers to decrease the amount of information that can be sent in timing channels. In [6], He et. al. discuss the presence of theoretical bounds on the amount of chaff needed to keep a timing channel from being distinguished from a Poisson process, if the packets in the "wheat" (not chaff) channel can be rearranged subject to a bound on delay before being sent. However, this approach requires committing to a certain constant rate of data exchange between hosts all the time, and requests to send data past the amount that can be safely hidden in the chaff still expose timing information to the outside. Furthermore, if these requests are not satisfied, the slowdown in communication can be detected by processes *within* the system, and exploited as a resource exhaustion covert channel (the bandwidth resource exhaustion covert channel exists anyway, but attempting to hide information well in chaff makes it much easier to hit).

The presence of these channels out of the Flume system, however, does not negate the usefulness of a networking addition to Flume. Many Flume deployments, such as a cluster of computers handling different parts of serving a website, will not have to worry about external adversaries with access to sniffing the cluster's network, if the local area network is properly secured. Even if a Flume cluster is widely geographically distributed, many use cases would provide enough traffic between nodes that a traffic-shaping covert channel out of the system would be low-bandwidth and unreliable. Finally, some deployments of Flume may only have the goal of reducing the chances of an accidental information leak; such deployments would not need to worry about traffic-shaping covert channels; such channels would require significant code complexity to work at all, and would be extremely unlikely to arise accidentally.

Chapter 6

Evaluation

To evaluate the performance of the network modifications to Flume, I compared network benchmarking python scripts running under Flume to the same scripts using normal network sockets running without any kind of flow control or monitoring. I ran tests of the round trip time on an established connection, the time to set up and perform one round trip on a new connection, and the time to send 1 MB ($10^2$0 bytes) over an established connection. I ran these tests both with and without `ssh` tunneling to add encryption, both because the `ssh` tunnel is an optional feature of Flume, and to compare the performance of unencrypted Flume to that of another protocol that proxies network connections.

6.1 Test Setup

To test the network code, I installed OpenBSD with Flume on two computers: one AMD Athlon^{TM}MP 1900+ at 1.6 GHz with 1GB of RAM, which ran the server half of the benchmarks, and one AMD Athlon^{TM}at 1.22 GHz with 384 MB of RAM, which ran the client half. The benchmarks used the MIT Parallel and Distributed Operating Systems group's office network, during a low-traffic time. The network path between the two computers contains two ethernet switches, and no routers.

The benchmarks were run as a python script that optionally runs under Flume and uses Flume network calls instead of the standard system calls. Other than the

choice of network library and the particular addresses to use, running the benchmark script has identical functionality with or without Flume. Experimentation with the command-line `ping` utility suggests python imposes little additional time overhead to the benchmarks, as the average round trip time as measured by `ping` is 0.314 ms, which is comparable to the average round trip time as measured by the python benchmark for round trip time of 0.430ms.

The Flume reference monitors on the two machines were run with all possible logging turned off, to better mimic a production environment and improve performance.

6.2 Benchmarks

I measured three different network properties:

Latency The server set up a socket on a port and listened on it. The client connected to the server, and the server and client each sent the other 64 bytes to prime the connection. Then, 256 times (in series), the server started a timer and sent the client 64 bytes. Upon receipt of the bytes the client sent 64 bytes to the server. When the server received all 64 bytes, it stopped the timer, recorded the elapsed time, and started the next trial on the same connection.

Connection Setup Time The server set up a socket on a port and listened on it. 256 times, the client started a timer, connected to the socket, and sent 64 bytes. The server received the client's message, and sent it's own 64 byte message back to the client. Upon receipt, the client stopped the timer, recorded the elapsed time, closed the current connection, and began the next trial with a fresh connection.

Throughput The server set up a socket on a port and listened on it. The client connected to the server and sent 1MB of data to prime the connection. Then, 256 times, the server started a timer and received 1MB more information from the client on the same connection, then stopped the timer and recorded the information before starting the next trial.

I ran each of these benchmarks:

- Using native network primitives, without Flume
- Under Flume, using a point-to-point Flume socket with a blank label tuple, without setting Flume to use encryption.
- Using native network primitives, without Flume, over a `ssh` tunnel.
- Under Flume, using a point-to-point Flume socket with a blank label set, with Flume set to use `ssh` tunneling encryption.

6.3 Results and Discussion

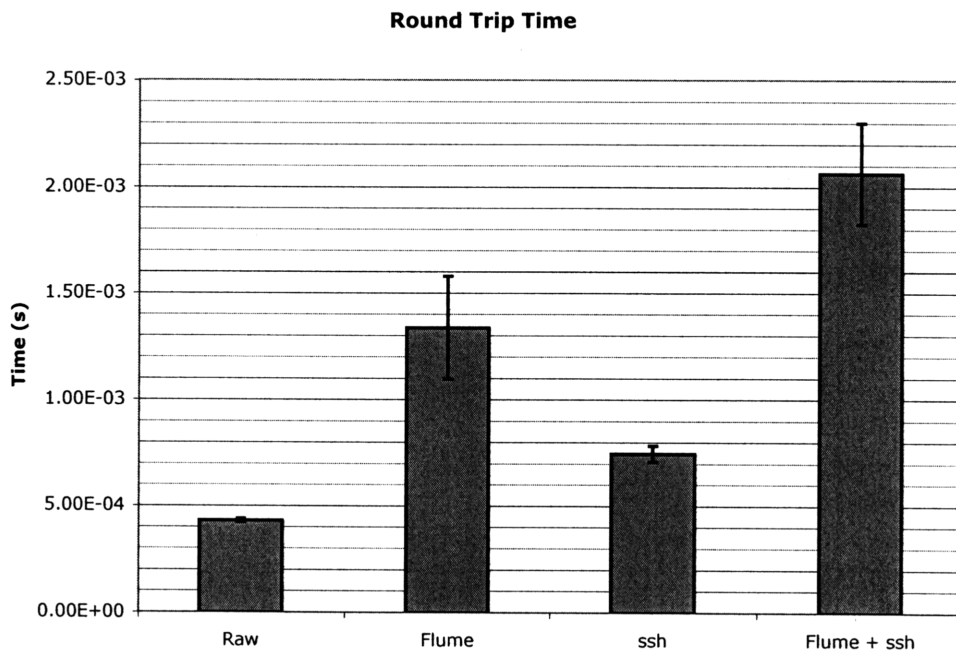


Figure 6-1: Round trip time comparison

The latency tests give the benchmark running under Flume a round trip time averaging 1.34 ms, which comes in at roughly a factor of three worse than the unconfined test without encryption at 0.430 ms. Comparatively, the unconfined test with `ssh` tunnel encryption has an average round trip time of 0.800, which is roughly a factor of two worse than the trial without the `ssh` tunnel. This result makes sense, as the `ssh` tunnel includes one proxy loop that must read from one computer and write to the network between the server and the client. Flume, however, contains two such proxy loops. If each proxy loop adds roughly another factor to the round trip time overhead, the results from this benchmark make qualitative sense. The average round trip time of Flume acting through an `ssh` tunnel is 2.06 ms, very close to what you would get if you simply added the times for the ssh round trip and the Flume round trip together.

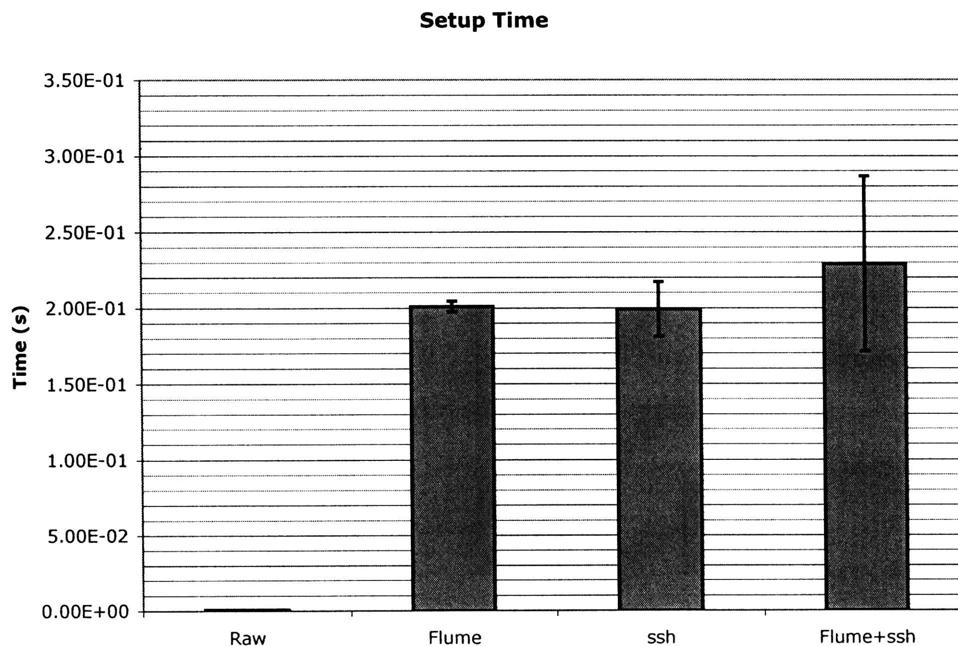


Figure 6-2: Setup time comparison

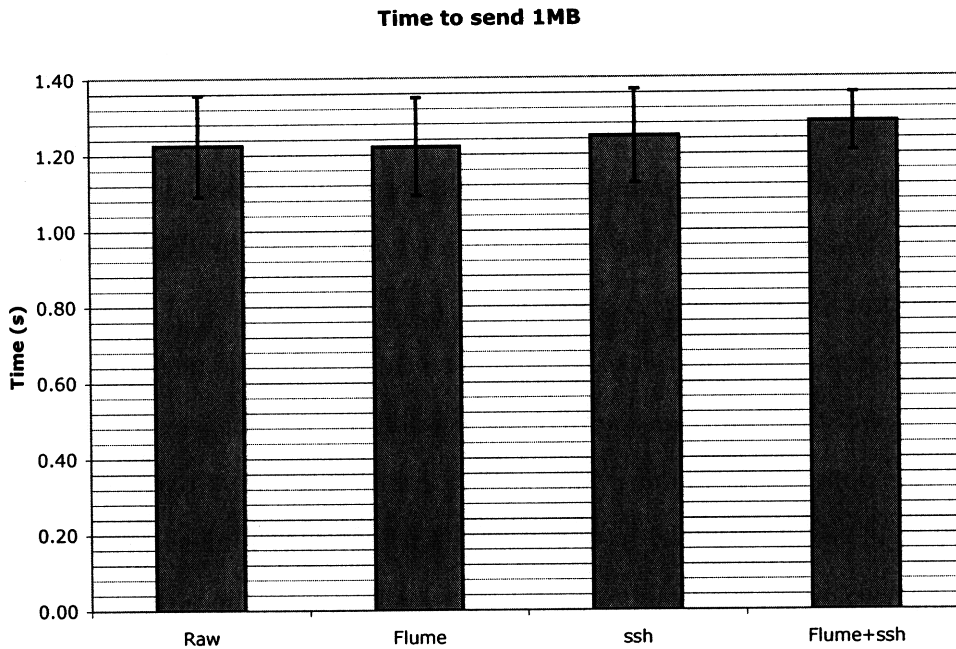


Figure 6-3: Throughput comparison

I predict that a wider or busier network between the client and server would decrease the effect of the round trip time overhead, as network transit time begins to dominate the time taken by the proxy loops.

The setup time tests show that Flume is currently unsuited for tasks where quick connection setups are important. The 201 ms average connection setup time for Flume is almost a factor of 250 worse than the equivalent operation without any encumbrance. Interestingly, the time for the benchmark to run under Flume, using `ssh`, and using both are comparable, to the point that it is difficult to say whether any is reliably faster than another. I am not entirely sure why `Flume` does not seem to pay the full extra cost for using the `ssh` tunnel, because it does internally make a new underlying connection for each Flume connection requested.

It is possible the setup cost could be reduced by making certain optimizations

in the reference monitor and network server. For example, setting up a connection currently requires a few requests to the IDD to look up the canonical versions of labels. If the results of these requests were cached, the setup cost might be significantly reduced.

Because of the long setup delay, along with the fact that the two-step rendezvous process can be onerous, I recommend that designers of clustered systems using Flume currently try to keep their connections between computers alive, instead of making a new connection for every new request.

The throughput tests give comparable performance for all the scenarios. Without using `ssh`, 1MB takes an average of 1.22 seconds to transfer, regardless of whether the program doing so is under Flume. With `ssh`, the time to transfer the data is only 0.02 (without Flume) to 0.06 seconds (with Flume) longer. This difference is less than the standard deviation of the results, which is on the order of magnitude of 0.1 for all the scenarios.

The good throughput performance suggests Flume is not going to hurt applications where the important factor in the clustered service is the throughput between machines. It also suggests that the buffer size used in the Flume proxy loops, 64KB, is large enough to avoid impacting performance.

Figures 6-1, 6-2, and 6-3 show the relative average performance of the three respective scenarios, with error bars corresponding to the standard deviation of the measured times.

Chapter 7

Conclusion

To make computer services that use Decentralized Information Flow Control techniques to increase their privacy and security more scalable, I have implemented an extension to the Flume reference monitor that allows programs on different computers to communicate in a way that preserves the information flow control properties of the system. Using this system, information does not need to be declassified before crossing the boundary between computers to another process running under Flume. The abstractions used by the extension to Flume are used in a similar way to most applications of POSIX stream sockets and should be relatively easy to use for people familiar with socket programming. Though connection and socket setup operations involving the Flume networking extension are expensive, the other operations impose a reasonably small amount of overhead on communication. Clustered systems that do not rely on fast connection setup are therefore reasonable to implement in a Flume environment for added information security.

Bibliography

[1] Stephen Adler. The Slashdot effect, an analysis of three internet publications. *Linux Gazette*, 38, March 1999. http://ldp.dvo.ru/LDP/LGNET/issue38/adler1.html.

[2] AFS user guide. http://www.openafs.org/pages/doc/UserGuide/auusg007.htm.

[3] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *SOSP '05: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 17–30, New York, NY, USA, 2005. ACM Press.

[4] Brad Fitzpatrick et al. Clustering livejournal, 2001. http://www.livejournal.com/misc/clusterlj/.

[5] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Symposium on Operating Systems Principles*, pages 78–91, 1997.

[6] Ting He, Parvathinathan Venkitasubramaniam, and Lang Tong. Packet scheduling against stepping-stone attacks with chaff. In *Proceedings of IEEE MILCOM*. Cornell University, October 2006.

[7] Maxwell Krohn, Alex Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *SOSP '07: Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles*. ACM Press, 2007. forthcoming.

[8] MoinMoin HelpOnAccessControlLists
. http://moinmoin.wikiwikiweb.de/HelpOnAccessControlLists.

[9] Robert T. Morris. A weakness in the 4.2BSD UNIX TCP/IP software. 1984. AT&T Bell Laboratories.

[10] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy*, 1998.

[11] Norman E. Proctor and Peter G. Neumann. Architectural implications of covert channels. In *Proceedings of the Fifteenth National Computer Security Conference*, pages 28–43, October 1992.

[12] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, January 2003.

[13] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63:1278–1308, 1975.

[14] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux security module. 2001. NAI Labs report #01-043.

[15] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: secure program partitioning. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*. ACM Press, October 2001.

[16] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, November 2006.