411 on Scalable Password Service

Federico D. Sacerdoti, Mason J. Katz,
Phillip M. Papadopoulos
San Diego Supercomputer Center
{fds, mjk, phil}@sdsc.edu

Abstract

In this paper we present 411, a password distribution system for high performance environments that provides security and scalability. We show that existing solutions such as NIS and Kerberos do not provide sufficient performance in large, tightly coupled systems such as computational clusters. Unlike existing single-signon services, the 411 design removes the need for communication during password lookup by using aggressive replication techniques. We demonstrate the use of shared keys to efficiently protect user information, and the careful management of system wide consistency and fault tolerance. A theoretical analysis of the behavior of 411 is matched with quantitative evidence of its performance and suitability to a clustered environment. We further show the system effectively responds to stress by simulating 50% message loss on a 60-node cluster. This protocol is currently used worldwide in hundreds of Rocks-based production systems to provide password and login information service.

1. Introduction

Distributing sensitive information such as passwords across a set of N connected nodes is a fundamental need for any coupled set of machines. Existing methods are often insecure, not scalable as N grows large, or both. While the security issue is obvious and pervasive, the performance shortfall is particularly evident in computational clusters, where many machines wish to obtain login information approximately in parallel. The goal of the 411 system presented in this paper is to provide security in this performance-sensitive environment.

NIS [7, 11] is the oldest and most established protocol for serving UNIX passwords. Despite being the de facto standard it has two important drawbacks, security and scalability. Security is addressed in NIS derivatives and by the mature Kerberos protocol [9]. Scalability is a more subtle but critical problem for this work. NIS is a centralized protocol where each lookup involves a query to a master node. This design works acceptably in loose-knit environments such as among workstations in an organization, but breaks down under more coordinated login behavior.

Since a cluster task is typically executed on a different node than it is launched, the startup process involves one or more queries for user credentials. A parallel application started on a cluster will therefore cause nodes to perform a storm of lookups for passwords and other user-specific information in a short, or nearly instantaneous timeframe. A parallel job startup in effect resembles a denial of service attack on a NIS master, with worse effects for a secure service due to its per-query encryption burden. In predictable manner, a centralized protocol cannot cope with such an attack, leading to a crashed or unresponsive service as many administrators have found first hand.

In this paper we present a novel system called 411 that provides secure and scalable password service to a tightly coupled set of nodes. The 411 protocol as described here has served for several years as the primary password distribution system in the Rocks cluster distribution [4] and is used on hundreds of computational clusters worldwide [10]. While intended for passwords, we show that 411's robust structure is applicable to a wide range of files.

To overcome the performance bottleneck in large systems, 411 aggressively replicates files on all nodes. Replication addresses the denial of service pattern of parallel queries because lookups are satisfied locally, without communication with a central server. This file-based technique moves the service burden from lookup time to data-update time, which is more tractable from a scalability standpoint. Our protocol secures data by encrypting files with a shared key on all nodes. This lightweight method eschews computationally-intensive secure tunnels for an *encrypt-once* strategy similar to sending an encrypted email to a mailing list.

From a correctness standpoint, our file-based system must carefully manage data consistency, and while not susceptible to lookup storms, 411 is somewhat affected by file size, which we examine in the experimental section. Since the absence of a password file can leave a node unusable, fault tolerance is of primary concern throughout the design and analysis of the system.

Given a defined security model we analyze the 411 protocol's resistance to a strong malicious attacker, beginning with the key distribution process and covering each protocol transition. Using formal techniques we prove 411 correct in the face of specific faults, a result buttressed by experiments of the protocol in action, where we force failures to occur and observe its recovery. We further present quantitative evidence that 411 provides data consistency on par with existing systems, and performance with large files.

The rest of this document is organized as follows. In Section 2 we examine related systems and protocols. Section 3 presents the goals and threat models used in our work, followed by the 411 design, including the key distribution system. Section 4 analyses our design from both a security and fault tolerance standpoint. Section 5 outlines some usability features of the protocol, and in Section 6 we present the experimental setup, and a discussion of the results. We give areas of future work in Section 7, and Section 8 concludes our paper.

1.1. Performance Requirements

Before presenting related work we further characterize the performance requirements of our environment.

Computational clusters often run either parallel jobs or "bag of tasks" workloads. Both stress popular password distribution systems in novel ways. Parallel jobs incur a denial of service query pattern on a centralized password system, due to the rapid and coordinated requests for user credentials from many nodes (figure 1). While enforcing a staggered job startup would ameliorate this problem, we do not want to impose such restrictions on the application or the job-launcher.

Bag of tasks workloads involve many independent calculations that run on a single or small number of nodes. Their task components are often short lived but numerous, perhaps a parameter sweep of a simulator. In this case as soon as one task completes, the next will start, perhaps on a different node. On a large busy cluster we can imagine many users running concurrently, while the scheduling system maps each new job to an available node.

In this example, not even a friendly staggered job-launcher can aid the password system. Together, the tasks may generate a torrent of user-credential requests, and being independent cannot coordinate or consolidate their query pattern. Asking the scheduler to space task assignments in time is wasteful.

Because of these usage patterns we hold that any secure password system in such a tightly coupled environment must provide robust and scalable service natively, without timing restrictions on its use.

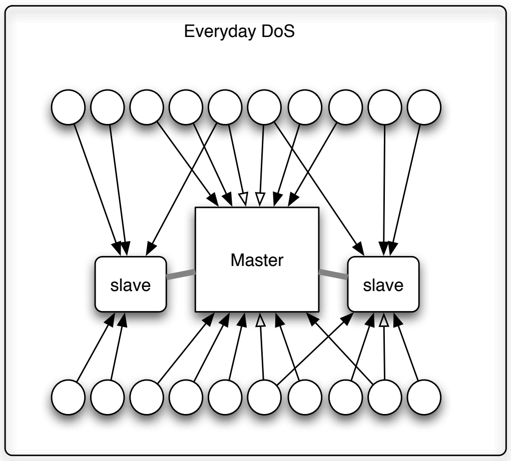

Figure 1. Parallel login behavior in a cluster resembles a denial of service attack for a centralized password system. Some queries may fail (empty arrows), generating more traffic.

2. Related Work

This section examines the most prevalent existing strategies for password service in a network. The two approaches most often used are centralized designs, those that employ replicated file databases.

2.1. Centralized Systems

All centralized protocols are susceptible to lookup storms as defined above. We identify popular systems that share this limitation.

NIS. The Network Information Service, once YP, is the *de facto* standard for password propagation. NIS centrally serves a text file such as /etc/passwd by hashing its lines into a *map*. Clients contact a NIS master whenever they need user information, providing a file-specific *key* and receiving the value in *map[key]*. While it is simple to setup and has excellent support in Unix and Linux, its shortcomings are security and scalability. NIS has weak security: it is easy make clients believe in a fake master [6] and password tokens are clearly identifiable across the network, making them vulnerable to a dictionary attack. However the system interface of NIS is quite useful, and has been copied in 411.

Slave NIS masters are designed to aid scalability of the protocol by mirroring the full database of the prime master. However experience has shown that even slave masters do not provide sufficient performance in a reasonably sized cluster [2]. Since slaves synchronize with the master in terms of whole files, the limit where every node is a NIS slave is essentially a basic file-replication strategy. Consistency between masters is maintained by cron job run every few hours.

NIS+ [12] is an extension to NIS. It adds much better security and some fault tolerance, but the protocol's propriety design has precluded its widespread adoption.

LDAP [14] is distributed service for user information storage and retrieval. It is structured as a general hierarchical database for any *mostly-read* data, and is designed to work across organizational boundaries. Recent implementations such as OpenLDAP [16] provide security via TLS tunneling. LDAP as a protocol is also not known for its performance [15].

Kerberos is a mature and secure protocol for user authentication. While difficult to setup, its security structures have been widely tested and proven. Although the service burden is more complex than NIS, Kerberos always requires a query-response between a client and server at login time and therefore we classify it as a centralized system. Kerberos allows for slave servers like NIS. Master and slaves synchronize with a periodic cron job, often at 15min intervals [8].

2.2. Replicated Databases

Rsync [1] is a popular and efficient file synchronization tool that uses hash functions to keep files updated by only sending differences to clients. Like 411, Rsync could be used to replicate the password file on all clients, and can support encryption via ssl tunnels. The protocol cannot effectively maintain encrypted files since its difference method will face a large edit distance for every file change. Rsync also has no automatic alert mechanism.

CFengine [18] is a rich configuration system for a set of heterogeneous nodes. It can distribute files to many clients, and can be directed to update files with a master server. However CFengine does not provide encryption, nor is optimized to keep a file consistent between nodes.

3. Design

This section gives the design goals and assumptions, followed by the relevant structures of the 411 design.

3.1. Goals

411 seeks to provide sensitive user information to a potentially large set of nodes that may make queries all at once. This mandate leads to several design considerations.

Security. 411 will distribute sensitive information that directly affects the security of the cluster. It therefore must be secure against common and reasonable attacks.

Scaling, Performance. The system must be scalable under lookup load: it may see N near-simultaneous lookups, where N is large. At no stage in the protocol may any node overwhelm its processing capability, regardless of the number of clients N.

Consistency, Usability Nodes must be able to retrieve reasonably consistent and accurate user information, given the rate of change typical for this type of data. In addition, the system must be at least as usable and easy to administer as current alternatives.

3.2. Threat Model

The 411 service is run on a strongly connected graph, where a node can reach any other node. We assume an adversary that can listen to all traffic on the network and inject arbitrary messages. All benign nodes operate correctly, and do not mistakenly send badly formed messages.

Our security goal is to prevent an adversary from discovering the plaintext contents of any 411 message, and to protect against denial of service attacks on the master node. Finally, client nodes can become malicious, leading to degraded but not catastrophic security guarantees.

The network that connects nodes can lose messages, as one would expect from a stressed packet switched network [18]. This may be viewed as another type of adversary that models the best effort delivery behavior of a standard IP network. Specifically, packet loss is message-oblivious [5], and we assume a random uniform loss of some percentage of all sent messages. This will affect analysis of our UDP signaling.

3.3. Performance

To escape the possibility of a master node bottleneck, 411 aggressively replicates login files. 411 deposits a copy of the password file on each client, so every node always has the most current version on its local disk. Therefore lookups of user credentials during a storm are embarrassingly parallel: all nodes consult their local password file without any communication with the master node.

In addition to replication for lookup scalability, 411 must provide encryption for security. The method used by OpenLDAP and Rsync is to tunnel all queries through secure SSL/TLS channels [20].

SSL/TLS transfers are computationally intensive, requiring up to ten times as much processing for a given flow as a plain TCP session [19],. In our preliminary implementation this overhead was so dramatic that using HTTPS as the transport mechanism in 411 was quickly rejected. We chose an alternative method based on plain HTTP transfers.

We could have chosen a multicast tree to distribute the files themselves, to save the cost of moving an identical file across the network to each client. However we rejected this transport mechanism for several reasons. First the simplicity and robustness of HTTP implementations allowed us to reason more effectively about the correctness and fault tolerance of our protocol. Second, we desire to extend 411 to the wide area in the future where a reliable multicast tree, while not impossible, would be a liability [28,29].

3.3.1. *Encrypt Once*

411 differs from other methods in that it globally encrypts data rather than using per-session secure tunnels. Imagine if members of a large mailing list all shared a private key. We could protect a message sent to this list by encrypting it once with our copy of the key, then sending the ciphertext through the plain mail system. 411 uses the same strategy. All nodes share a key, and the master node encrypts files once per version instead of once per version-client as tunneling requires.

In addition to the 256-bit shared key for symmetrical file encryption, the master keeps a 1024-bit RSA key pair for signatures, the public key of which is held by clients. A 411 file is encrypted with the shared key, signed with the master's private RSA key, and served via plain http to all clients that request it. It is the client's responsibility to decrypt and verify the file after transfer. Therefore the burden of a file's encryption is well distributed, *O(1)* for all nodes. If we instead used secure tunneling, the burden would be *O(1)* for clients and *O(N)* for the master.

The observation is that since we always transfer the whole database, responses from the master to client queries are the same for a given file version. As the plaintext is unchanged, we only encrypt it once. This strategy could be applied to existing master-slave synchronization protocols used by Kerberos and NIS+ as well.

3.3.2. *Alerts*

To reduce inconsistency, we signal all clients on a file change. When a file on the master is altered, the master sends an *alert* multicast to the cluster containing a URL to the changed file. On receipt of an alert, a 411 client initiates an http transfer (figure 2). Alerts are sent in Ganglia compatible format [21,27] and can be monitored and tracked like any other metric (figure 4). Like other protocols in Rocks [18], 411 uses multicast signals so a master does not need *a priori* knowledge of its clients: only possession of the shared key delineates cluster members.

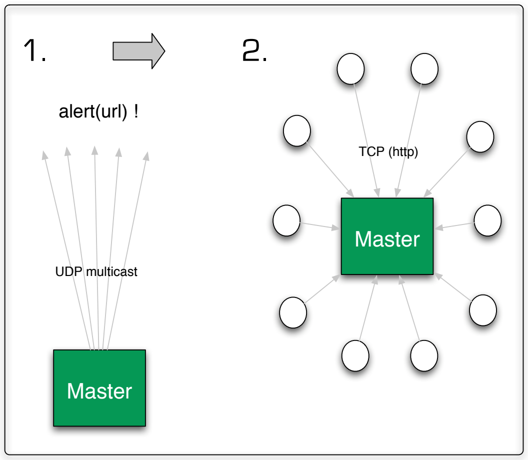

Figure 2. 411 design. When a file has changed, the master constructs, signs, and multicasts an alert for it (1). On receipt, clients verify the signature, and retrieve the file from the master via http (2).

True Denial of Service attacks are easiest and most effective when a small effort on the part of the attacker leads to a large response in the system. Forging or replaying a stored 411 alert message would easily affect a large response: all clients would immediately ask for the new file from the master.

To protect against this risk, all alerts from a master are signed to prove their authenticity. To prevent replay attacks, alerts contain a strictly increasing number (currently a timestamp). This token and the URL are signed with the shared key, which forms the whole of the alert (figure 4). Clients ignore all alerts that do not verify, and keep a record of the latest timestamp received from every master, and previously seen are ignored. Finally, we know the URL and master address is trustworthy because a valid signature protects it.

As a scalability optimization, clients randomly stagger their retrievals on an alert receipt. Clients use heartbeats from ganglia agents on all nodes to establish a size estimate of the cluster. This estimate is used to calibrate a small random backoff timer. On receipt of an alert, a client backs off slightly before retrieving the file in order to stagger the load on the master server. This timer is set at *random(0, N/100)* in our implementation, where *N* is the size estimate.

3.4. Initial Key Distribution

The initial shared key distribution is not handled by this protocol. While the shared key could be manually

```
-----BEGIN 411 MESSAGE-----
wkkZ4x8PNg36ZOZwXJEBIdhGOdVES4zRVX+h+ip0ruGVH8Te9SK/bEdXX7aCVk1+sVeW8CtOasL7
JXDrFa6LE+Z4NLuPRsjw8HUrMDUIUC+862huRe6tI2lT9IwDojcm5Lif7GXTkh9nMhaLJgvFW31b
4pQVEltE1mTYYru3Tv0=

jEEE3LLhKQ/rBffQoFmC249skFaEh+6ovRzMii2/Oqxe2A5c2z4fXkzNLPdU1QxdBRkrXh5mvgfG
lWiVWiUTcm8C2jzzoJAhNBvTJ2h3Lw7g6wuA7TNhCaCsM8iPMqzQ8TqzUM22eAUumed0U17i/Wif
Vi3OtBDlb98Iapx4xbBkYC9O6ICCFMGjTjsk4rkTc0AZOhxR6oHgY+PpntOQSi+SQ4Trr14XoUAw
sLtOhemqBc5JJb6kOkmXMVnr/jEJxzx9WWhRvYGsUM8x42dE4neC28jxaXl5TtrO4ywjkcTAR/KU
7dXpsNBlPRbEf0XCGzM8elzaaTfHyLfJwsCoVxmIWdSymF5HVUc+2lUfpfMIlAhVTqCWJ1qZbIB4
...
s/tKuNCZXsknBsToxIUceOSj2INgkXfcHeg0iVoBCOMuy3rZxj8bDEXIbT+OsMYVhqD6tqmA+bB6
-----END 411 MESSAGE-----
```

```
# $411id: /etc/passwd$
# Retrieved: 24-Jan-2005 15:40
# Master server: 192.168.69.1
# Last modified on master: 24-Jan-2005 13:24
# Encrypted file size: 2.4K bytes
#
# Owner: 0.0
# Name: /etc/passwd
# Mode: 0100664
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
...
```

Figure 3. A password file in encrypted (top) and decrypted form (bottom). Master serves encrypted file over http. Clients perform decryption. 411 headers for meta-data are visible in the comments of the decrypted file. The encrypted form shows the signature and ciphertext paragraphs.

```
<METRIC NAME="411alert" VAL="http://10.1.1.1/411.d/etc.passwd 1106614104.16
oC0rG9r07IOxOO4o/1YY79Mim8Sm7oPTWnfPzOoaoEJhTxySl3vvvrbsyv2iKe9SViCQIgvSLMeS
vA3Ccqcqa27fDe4Y8c8EpLXWCRTHKPXwRcmyWM6VLPRHWEzp1/MQ7eHmYKqJxt0wZl7X/ng2Ws/J
K7dEyeswkQEgivWMj/Y=
" TYPE="string" UNITS="" TN="14" TMAX="20" DMAX="20" SLOPE="zero"
SOURCE="gmetric"/>
```

Figure 4. A 411 alert multicast message in Ganglia XML format, telling all recipients the file 'etc.passwd' has changed on this master. The java-style URI is to prevent naming collisions only, meta-data is encrypted in the 411 file itself. The URL and timestamp are signed to prevent replay and forgery attacks.

placed on clients using a secure file transfer tool, such a method relies on the absence of human mistakes during the process. We prefer to automatically place the shared key during node installation using the Rocks system.

Rocks employs a bimodal security structure to control node discovery. When adding new nodes, a human places the frontend into an insecure mode where it will serve entrance requests from any node. In this phase such a request will yield a kickstart file containing initialization instructions along with the 411 keys, secured with SSL/TLS. Once new nodes have been discovered, the frontend is returned to a safe mode. Now only registered nodes can obtain the kickstart file as proved by a signed certificate given during the initial installation.

The standard practice is to partition the frontend from the public network before placing it into discovery mode. This strategy relies on a temporary trust that your local-area network is free from attack during this phase. We plan to augment this temporal strategy with a physical USB-key method in the future.

3.4.1. File Meta Data

Before encryption, the master adds several HTTP-style headers to the file describing its fully-qualified name, permissions, type, and owner/group ids. This information is encrypted along with the file's contents to provide a secure, trusted record of its meta-data. Like HTTP, 411 allows implementations to add custom headers as needed.

3.5. Protocol

411 (informal): When an interesting file changes, the master node encrypts it with the shared key, signs it with the master private key, and stores it on the server. The master then constructs an alert and sends a series of copies using either UDP multicast[1] or normal IP broadcast. On receipt of an alert, clients retrieve the file if appropriate. On a polling event clients retrieve all files available from their favorite master.

411 (formal): Pseudo I/O Automata format

```
States_i:
411Files, a set of filenames, non-empty on master.
Stamps, a hash table of real numbers indexed by url.
Scores, a hash tbl of saturating counters, indexed by ip.
R, a queue of alerts. (all initially ∅)

Transitions_i:
changed(f), f ∈ 411Files:            Input
        encrypt file with shared key, sign with RSA-priv
        send alert to all i

recv-alert(a_j):                     Input
        verify signature
        if Stamps[url] ≥ ts(a): return
        if i = j (we sent the alert):
                add a to repeat queue R
                return
        else:
                get(url(a))
                Stamps[url] = ts(a)
                increment Score[ip(a)]

repeater:                            Output
        Enabled on a random periodic interval with
        expectation 10s.
        ∀ a ∈ R:
                send a to all i
                remove a from R if older than 1 hour

poller:                              Output
        Enabled on a random periodic interval with
        expectation 5hrs.
        m = master with highest score
        ∀ f ∈  List(m):
                get(url(f))
```

[1] Multicast overlays may be used to extend the alert reach to the wide area, however such use is not examined here.

The internal method get(url) retrieves file at url, decrypts it and verifies the signature, then writes file to local disk. If an error occurred, get() decrements the master's score, then simply causes the action to end.

4. Analysis

In this section, we explore the methods used to secure information in 411, and analyze its fault tolerant and consistency properties.

4.1. Attacks and Defense

Our design can withstand various types of attacks on the system. We note that innocuous activity such as interference from other valid clusters may cause attack patterns as well.

Potential security attacks can come in three forms: compromised keys, intercepted transfers, and injected traffic. The automatic nature of our system requires that no passwords be manually entered, therefore only filesystem permissions protect the shared key. We also consider the case where a client node has been compromised and itself becomes a malicious agent.

Guessing the 1024-bit RSA master private key, which signs all files and alerts, is computationally infeasible by the Diffie-Hellman assumption [22]. Guessing the 256-bit shared key is even harder, since no public key is available to guide the search. We feel our use of cryptographically safe random number generators [23,25] offer sufficient security against guessed keys and do not require further analysis here.

Intercepted http transfers do not reveal information because all file contents is encrypted as a unit. Although passwords are always kept encrypted, we prevent dictionary attacks by making it infeasible to delineate passwords tokens in a 411 transfer stream.

Injecting packets into the network is another possible attack. Imagine a malicious attacker Darth. Darth cannot forge viable alerts since he lacks the master private key. Since clients ignore all alerts without a "newer" timestamp than they have seen before, Darth gains little by replaying old but valid alerts. The most he can hope for is to fool nodes that have no state because of crashes, etc. The master sends alerts periodically to catch-up recovering nodes.

If Darth convinces the Rocks frontend that he is a valid cluster member he will obtain the shared key. While such a malicious client can obtain the plaintext of 411 files, without the master key, the more serious case of subverting the entire cluster with a believable password file is impossible. Defense against a compromised master, however, is outside the scope of our model.

4.2. Consistency

The 411 system is a replicated database of files. The replication strategy yields fundamental scalability, but relies on fault-tolerant progress towards consistency after updates. If a login file changes, it takes some time δ to successfully deliver the new version to clients. During δ the system is not consistent: some nodes may have the newest version, some not.

Existing protocols with slave servers such as NIS and Kerberos also allow periods of inconsistency when the master has more recent information than slaves. Both systems perform the synchronization operation using a cron-like periodic process with a typical interval on the order of minutes [8,11,12]. We would like 411 to maintain at least this level of consistency across the cluster.

Instead of relying on a periodic process, we use multicast UDP messages to speed consistency progress during transitions. However UDP broadcast is not reliable, and the chance of an error goes up with cluster size. The 411 master therefore repeats new alerts on the network periodically. As a final safeguard to ensure eventual consistency, each client maintains a *polling* agent that retrieves all available files from the master at a low frequency interval, currently 5 hours.

4.2.1. Fault Tolerance

We would like to show 411 guarantees all nodes will receive the current version of a file with an overwhelmingly high probability, even when subjected to faults from our model. Changes always originate on the master node; just after a change to a 411 file f, all clients are in the *inconsistent set* which we will call I_f. When a node successfully receives the file f we remove it from I_f and add it to the consistent set C_f. Perfect consistency occurs when set I_f is $\varnothing$.

The sets are disjoint, so $I \cap C = \varnothing$. The argument for consistency under faults is in two parts:

A. All nodes eventually receive the broadcasted alert for f

B. All nodes eventually get the file and decrypt it, increasing the set of consistent nodes.

To show (A), we note the 411 master repeats alerts frequently to overcome the inherent unreliability of the UDP multicast medium. This strategy, shared by the Ganglia protocol, ensures that with a high probability all nodes eventually receive the alert for f. Since the possibility of loosing an alert is identical for all repeats (by our network loss model), we may use the binomial distribution, shown as *b(x)* below, to predict the chance a single node receives at least one alert. The random variable X is the number of received alerts on a node:

$$P_{one}[X \geq 1] = 1 - b(1), \qquad (1)$$

$$b(1) = \binom{n}{1} \; p(1-p)^{n-1}$$

where n is the number of alerts sent, and p is the probability of receiving a single alert. The probability of all nodes receiving at least one alert is P_{one} raised by the number of clients N:

$$P_{all}[X \geq 1] = (P_{one}[X \geq 1])^N \qquad (2)$$

In our implementation repeats are sent at a random interval with an expectation of 10 seconds. The repeater task removes any alert older than one hour, so the average number of alerts sent per file version is 360. Every alert fits into one Ethernet packet, so the real chance of loss is low for most networks. However the equations above predict 411 will deliver at least one alert to all nodes even when subject to very high loss.

On a 60 node cluster, even if 50% of all sent alerts are lost, this model predicts we need only 30 repeats to be 99.99999% (five nines) sure all nodes receive at least one alert. We observe by (2) that larger clusters need more repeats, but our approximately 360 alerts are mathematically adequate for thousands of nodes.

While we are confident all nodes will receive the alert signal, to show (B) we must consider failures in the HTTP GET pathway. We expect faults during the http file retrieval phase, since requests come in bursts and our web server enforces finite resource limits. If apache allows only n outstanding requests, the $n+1^{st}$ will get no response, perhaps timing out. To handle this case we use our alert fault-recovery mechanism for two purposes.

Clients only update their timestamps on a retrieval success: any failure in the file transfer is treated like no alert was received at all. The client will try again on the next alert that arrives, effectively using the alert stream as an end-to-end failure recovery mechanism (figure 2). Therefore by our design if (A) holds (B) will hold as well.

We have predicted the fault tolerant mechanisms of 411 are sufficient for correct operation to our standards. In the experimental section we provide quantitative evidence of this result.

5. Experiment

Our quantitative examination of 411 seeks to establish claims predicted in the analysis. First that all nodes reach a consistent state when subject to alert (UDP) and get (TCP) failures. Second that uptake time (the inconsistent window after a change on the master) is reasonable given the random behavior of the alert repeater. Finally we establish reasonable operation when we increase the size of the login files.

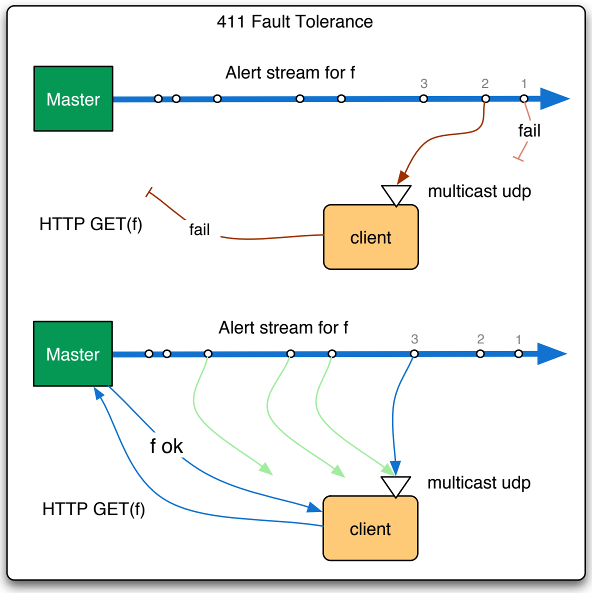

Figure 5. 411 fault tolerance. When a file *f* changes, the master sends a stream of identical alerts on the multicast channel. We see two types of failures (top): the first alert is lost in the UDP medium, the second leads to a failed file retrieval. On the third alert the client successfully retrieves *f* (bottom), and ignores the remaining alerts.

While the 411 poller task is present for final fault-tolerance in the system, all pollers are disabled during these experiments.

5.1. Setup

We run the experiments on 61 nodes of the Rockstar cluster [26]. Nodes are approximately homogenous, with dual 2.8Ghz P4 processors and 2GB RAM. The network consists of a Gigabit Ethernet tree with two 48-port switches connected via a 4-port trunk. The cluster runs the Rocks software version 3.3.0. The 411 service is configured with the cluster frontend serving as the single master for the 60 clients.

We measure 411 file retrievals using a ganglia metric sent from clients on a success. These signals are measured on the frontend by an agent listening directly to the ganglia multicast channel. While these signals are themselves subject to failure, we treat such faults as indistinguishable from faults in the 411 processes themselves.

Other than the operation of 411 and several standard Linux daemons, no computationally significant tasks were run during the experiments.

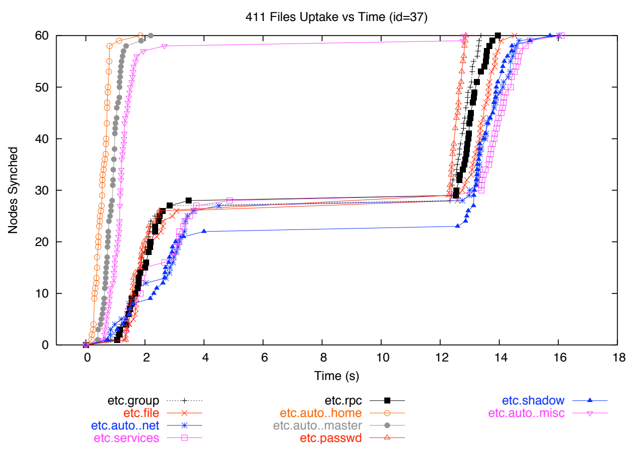

Figure 6. File synchronization in 411. At time zero, ten files are changed and alerts broadcasted. Resource limits on the master HTTP server cause failures visible in the flat curves after 4s. The repeated alert that arrives in the 12th second effectively picks up the remaining clients. This graph shows the short response time of our protocol and its tolerance of HTTP failures.

5.1.1. *Operation under Faults*

In this experiment we change multiple files on the master and plot when nodes retrieve each file. Each client requests 10 files from the frontend web server, which translates to approximately 600 requests in under 1 second (figure 5). We expect to see the web server enforce its resource limits.

In this process we mark time zero, then touch 10 login files listed in the key of figure 5. We then run the 411 Makefile, which encrypts and sends an alert for each file sequentially. This sequence of actions mimics adding a real user to the cluster but with more files changed.

5.1.2. *Average Uptake Time*

Here we characterize the uptake time of 411 over many trials. We repeat the first experiment several hundred times and how long until all nodes have successfully synchronized. Since the alert stream is the bottleneck to completion, we show the results in a histogram to characterize its random behavior.

Figure 6 gives the histogram for four cluster sizes, ranging from 8 to 60 nodes. Two hundred uptake cycles were run per size. We introduced no message loss during this experiment, other than what naturally occurred in the network.

5.1.3. *Uptake Time with Message Loss*

To correlate actual behavior with our predicted performance we artificially lose messages here according to our network model [18]. Clients simulate alert loss by flipping a random uniform coin every time an alert message arrives. If the coin is tails, the message is “lost” and the client takes no action. On heads the clients behave normally.

Since the network itself may lose messages we note that the loss in the system is at least 50% and perhaps more. Figure 7 presents a histogram of the results as in the previous experiment, but with 30-second buckets to better show the longer and more varied times.

5.1.4. *Large Files*

In this experiment we measure 411’s ability to synchronize when the changed file is large. Unlike previous runs here we only change a single file on the master. The file sizes range from several kilobytes (typical for a password file), to ten megabytes as a pathological case (figure 8).

5.2. Discussion

The results of the first experiment show the fault tolerant mechanisms in action. Figure 6 shows the burst of http accesses that force the web server to invoke its resource limits. The Makefile sends alerts sequentially, leading the first files (auto.home, auto.master,) to finish contiguously as the 60 nodes can fully retrieve several files immediately. The remaining files must wait for another alert to finish their synchronization.

The Apache server on the master node allows no more than 150 concurrent unfinished accesses at a time. After this limit nodes begin to experience failure as seen in the flat curves of figure 6. We observe partial success in this first phase as early clients finish their transfer and free resources, but clearly the alert that arrives in the 12th second leads to important retries. By 16 elapsed seconds all files have been successfully retrieved and the cluster is consistent.

The second experiment shows the distribution of many such runs. We know from the first experiment the repeat stream is the bottleneck to completion. The alert task on the master is calibrated to send repeat alerts at a random uniform frequency with an expectation of 10 seconds. Figure 7 shows a normal distribution with a mean of ten seconds as we would expect, with a tail to 130 seconds. The smaller clusters often finish completely on the first alert, as is evident by the taller first bar representing synchronization within 0-10 seconds.

At worst nodes take three minutes to fully synchronize. This compares well with NIS and Kerberos, which commonly synchronize slaves with a 15 to 60min cron job. The inconsistent interval in 411 is not trivial, however, and has implications for dynamic, on-demand user creation.

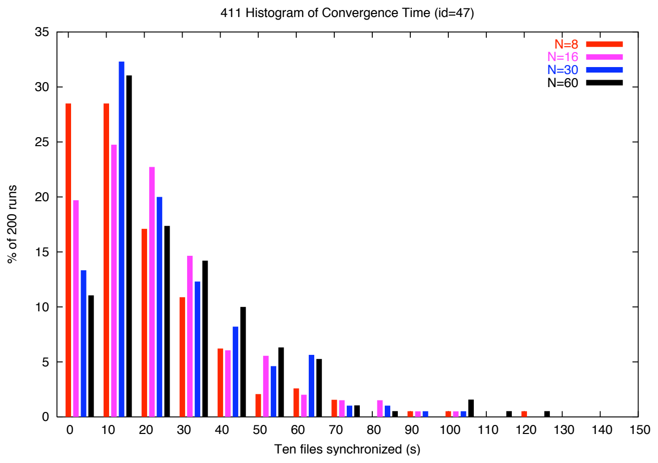

Figure 7. Histogram of uptake time. Ten files are changed and synchronized as in figure 6, but process is repeated 200 times. The bars labeled 10 show what percent of runs took between 10-20s to finish for each cluster size. We see a normal distribution with a mean of 10s as expected. This chart further supports our claim that 411 has acceptable response time.

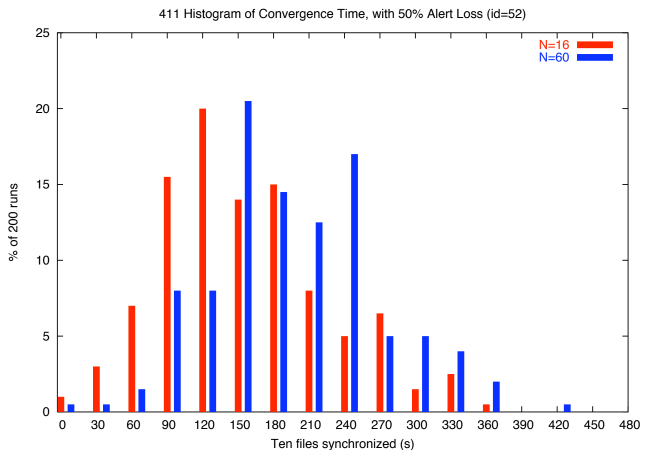

Figure 8. Histogram of uptake time with alert loss. Clients simulate message loss by randomly discarding 50% of received UDP alerts. A 16 and 60 node cluster synchronize after the master changes ten login files as in figure 6. Uptakes are noticeably slower, but all are without error. This result shows the correct operation of our protocol under extreme stress.

In the third experiment 411 operates under stress from 50% alert message loss. Nodes take longer to finish in this case, with a mean of several minutes as seen in figure 8. However, all runs complete, and every client retrieves the files without failure.

In our analysis we predicted that 30 repeats would be necessary for all clients to receive at least one alert. We expect 30 alerts have been sent by time 300, and the results show a majority of clients have finished by this time. We attribute the fact that less than 99% have completed to HTTP failures that require clients to receive more than one alert.

The different peaks in figure 8 show that large clusters take longer to synchronize than small ones. While we never expect to see 50% message loss in a real cluster, the stress of this experiment illuminates its operation in a larger environment. The result shows that 411 uptake time is related to *N*. While this window may stretch beyond our five minute goal for very large clusters, we argue this experiment shows the protocol will continue to perform within useful bounds.

From the results of the fourth study it is clear that 411 on this hardware can achieve consistency of a single file in less than a minute regardless of its size. However the variation in uptake time (figure 9) even with our small cluster of 60 nodes is significant. We note that 411 may not be appropriate for synchronizing very large files across the cluster.

Finally, in the 10KB file (marked with 'x' points in fig 9) has a tail indicating some straggler nodes did not retrieve this file until two seconds after receiving the alert, well past the maximum backoff time of 0.6s. We attribute this tail, and the non-linear 10MB curve, to normal operating system variations on the master that cause the apache web server to occasionally respond more slowly than normal.

6. Usability

This section addresses how 411 is used in a real cluster.

6.1.1. System Interface

For the success of 411 we must also provide an interface to the larger system that is easy to use. Like NIS, we employ a Makefile that automatically generates encrypted versions of login files such as /etc/passwd, and /etc/shadow and alerts. On Rocks systems the *useradd* binary invokes this Makefile during account creation and modification. Therefore in large part system administrators do not know they are using 411 rather than NIS.

In real world use, there may be cases where the network has been partitioned for some time before being healed. In this case entire 411 alert streams may be lost and partitioned nodes will be inconsistent with the master. We have provided mechanisms to quickly force all alerts to be resent from the master, or have the poller run immediately on affected nodes. The ability to react to such types of complete network failures has proved a useful in practice.

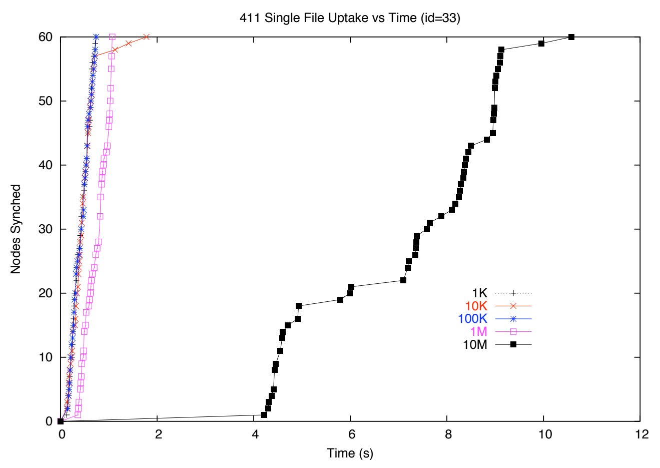

Figure 9. 411 uptake performance with various file sizes. At time zero a single file is changed on the master. When 60 nodes have retrieved the new version the cluster is consistent. A typical password file is 10KB; the 10M run simulates a reasonable worst case. The initial flatness of the 10M curve is the master's encryption time. This experiment illustrates how file size affects 411.

6.1.2. *Multiple Masters, Backoffs*

If multiple masters are present, 411 clients will automatically register them (via their alerts) and randomly choose one to poll from. Clients respond directly to alerts, as they are indications of an alive master. We do not provide a special mechanism to keep multiple masters synchronized, however a tree of masters has been used in practice.

Clients keep a saturating counter for each master server they know. This score counter represents the health of a master server and is incremented on a good GET, and decremented on failure. Clients use their local score counters to choose a master to poll from, enabling a steady movement to the most healthy master without manual intervention.

6.1.4. *Groups*

Groups are a facility to maintain a 411 file between the master and a subset of clients. Membership in a given group is determined by a configuration file on the client, and therefore is delineated by choice only. This convenience feature names groups with a path, such as */Compute/Rack1*, which is sent with every alert and is visible to pollers. Clients locally determine whether a file is interesting based on its group designation. Groups have inheritance semantic that full paths imply interest in their prefix. For example a compute node interested in group */Compute/Rack1* will also choose files in group */Compute*. The group facility is implemented as directories in the URL for easy listing and retrieval.

7. Limitations and Future Work

411 makes no effort to work well across administrative boundaries. Unlike systems such as LDAP, 411 currently has no provision for merging data from multiple master authorities.

While using a multicast overlay network to extend the alert reach is theoretically possible, we leave such efforts to future work.

8. Conclusion

The task of distributing sensitive information across a set of N nodes has no good solution in tightly coupled high performance environments. Existing methods are inherently not scalable, and strategies to add security with secure tunneling cannot provide sufficient performance due to their computational burden. In response to these shortcomings, we presented 411 as a secure, scalable, and fault tolerant password service.

We have shown how 411 removes the query time bottleneck from lookup storms with aggressive replication, efficiently distributes encryption load. We show that the protocol is correct and robust to failures, and remains strong under various types of malicious attacks. We presented evidence of our performance and scalability claims on a 60-node cluster. We demonstrate that 411 maintains at least as much password consistency as existing systems we studied. Finally we characterized the expected performance of the system for various file and cluster sizes.

While this work is an early foray into the area of high performance security for password systems, the theoretical, experimental, and real world experience with 411 establishes it as a useful system for distributing user credentials to a set of nodes.

Acknowledgements

This paper has benefited from the insight and critique of many people. Thanks to Greg Bruno for his invaluable support and feedback, and to Matt Massie for his help with replay attacks. We would also like to thank the anonymous reviewers for their helpful comments. Most of all we must thank the countless Rocks users that struggled through early and less fault-tolerant versions of 411 on their systems.

All code necessary to re-create experiments in this paper is freely available upon request.

References

1. A. Tridgell, P. Mackerras. *The rsync algorithm*. Technical report, Australian National University. http://rsync.samba.org. 1998

2. A. Keller and A. Krawinkel. *Lessons Learned While Operating Two Large SCI Clusters*. In Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid (CC-GRID). Brisbane, Australia. 2001

3. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers. 1996.

4. P. Papadopoulos, M. Katz, G. Bruno. *NPACI Rocks: tools and techniques for easily deploying manageable Linux clusters*. Concurrency and Computation: Practice and Experience, 15:707-725, 2002.

5. B. Chor, M. Merrit, D. B. Shmoys. *Simple Constant-Time Consensus Protocols in Realistic Failure Models*. JACM, 36(3):591-614, 1989.

6. D. Hess, D. Safford, U. Pooch. *A Unix network protocol security study: Network Information Service*. Computer Communications Review, 22(5):24-28, October 1992.

7. Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

8. Sun Microsystems. *System Administration Guide: Security Services*. Sun Documentation #817-0365, August 2003.

9. J.T. Kohl and B.C. Neuman. *The Kerberos network authentication service (version 5)*. Internet Engineering Task Force RFC 1510, September 1993.

10. *The Rocks Cluster Register*. http://www.rocksclusters.org/rocks-register/

11. Sun Microsystems. *System Administration Guide: Naming and Directory Services (DNS, NIS, and LDAP)*. Sun Documentation #817-2655, December 2003.

12. Sun Microsystems. *System Administration Guide: Naming and Directory Services (NIS+)*. Sun Documentation #816-4558, January 2005.

13. J.T. Kohl and B.C. Neuman. *The Kerberos network authentication service (version 5)*. Internet Engineering Task Force RFC-1510, September 1993.

14. W. Yeong, T. Howes, and S. Kille. *Lightweight Directory Access Protocol*. Internet Engineering Task Force RFC 2251, December 1997.

15. S. Fritzgerald. I. Foster, C. Kesselman, G. von Laszewski, W. Smith and S. Tuecke. *A directory service for configuring high performance distributed computations*. In Proc. 6 th IEEE Symp. On High Performance Distributed Computing, August 1997.

16. The OpenLDAP project. http://www.openldap.org/

17. M. Burgess. *A site configuration engine*. USENIX Computing Systems, 8(2):309--337, 1995.

18. F. Sacerdoti, M. Katz, G. Bruno. *On Distributed Agreement for Clusters*. SDSC Technical Report TR-2004-1, 2004.

19. C. Coarfa, P. Druschel, and D. Wallach. *Performance Analysis of TLS Web Servers*. In Proceedings of NDSS, 2002.

20. T. Dierks and C. Allen. *The TLS Protocol, Version 1.0*. Internet Engineering Task Force RFC 2246, January 1999.

21. F. Sacerdoti, M. Katz, M. Massie, D. Culler. *Wide area cluster monitoring with ganglia*. In Proceedings of the IEEE Cluster, Hong Kong, 2003.

22. W. Diffie, M. Hellman. *New directions in cryptography*. IEEE Transactions on Information Theory, 22(6):644–654, 1976.

23. OpenSSL: The Open Source Toolkit for SSL/TLS. http://www.openssl.org.

24. J. Callas, L. Donnerhacke, H. Finney, R. Thayer. *OpenPGP Message Format*. Internet Engineering Task Force RFC 2440, November 1998.

25. The Python Cryptography Toolkit. http://sourceforge.net/projects/pycrypto

26. The Rockstar Cluster. http://rocks.npaci.edu/rocks-register/details.php?id=148

27. M. Massie, B. Chun, D. Culler. *The ganglia distributed monitoring system: Design, implementation, and experience*. 2003. Submitted for publication.

28. J. C. Lin and S. Paul. *RMTP: A reliable multicast transport protocol*. Proceedings of IEEE Infocom, pages 1414--1424, Mar. 1996.

29. B. Rajagopalan. *Reliability and Scaling Issues in Multicast Communication*. Proceedings of ACM SIGCOMM, 1992.