Multiprimary support for the Availability of Cluster-based Stateful Firewalls using FT-FW

P. Neira[1], R.M. Gasca[1], L. Lefèvre[2]

[1] QUIVIR Research Group - University of Sevilla, Spain, pneira, gasca@lsi.us.es
[2] INRIA RESO - University of Lyon, laurent.lefevre@inria.fr

Abstract. Many research has been done with regards to firewalls during the last decade. Specifically, the main research efforts have focused on improving the computational complexity of packet classification and ensuring the rule-set consistency. Nevertheless, other aspects such as fault-tolerance of stateful firewalls still remain open. Continued availability of firewalls has become a critical factor for companies and public administration. Classic fault-tolerant solutions based on redundancy and health checking mechanisms does not success to fulfil the requirements of stateful firewalls. In this work we detail FT-FW, a scalable software-based transparent flow failover mechanism for stateful firewalls, from the multiprimary perspective. Our solution is a reactive fault-tolerance approach at application level that has a negligible impact in terms of network latency. On top of this, quick recovery from failures and fast responses to clients are guaranteed. The solution is suitable for low cost off-the-shelf systems, it supports multiprimary workload sharing scenarios and no extra hardware is required [3].

1 Introduction

Firewalls have become crucial network elements to improve network security. Firewalls separate several network segments and enforce filtering policies which determine what packets are allowed to enter and leave the network. Filtering policies are defined by means of rule-sets, containing each rule a set of selectors that match packet fields and the action to be issued, such as accept or deny. There are many problems that firewalls have to face in modern networks:

1. **Rule set design**. Firewall rule languages tend to be very low level. Thus, writing a rule set is a very difficult task [1] and usually requires an in-depth knowledge of a particular firewalls' internal working. Furthermore, each vendor has its own firewall language. The research community is trying to construct a standard language to express rule-sets that compile as many specific low level languages as possible [2].
2. **Rule set consistency**. When rules are expressed using wildcards (i.e. filtering entire subnets instead of single IPs) then the rules may not be disjoint.

[3] This work has been partially supported by the Spanish Ministerio de Educación y Ciencia through a coordinated research project(grant DPI2006-15476-C02-00) and Feder (ERDF).

In such a situation rule ordering is important and it can introduce a consistency problem. Moreover, if on the route from the sender to the destination, multiple firewalls are crossed, a consistency problem can be introduced between the rule-sets of firewalls. Building a consistent inter-firewall and intra-firewall rule-set is a difficult task, and even more challenging if it must support frequent dynamic updates [3]. Also, several works have focused on solving consistency and conformity problems in rule-sets and also in distributed environments [4] [5] [6].

3. **Computational complexity**. As each packet must be checked against a list of ordered rules (or unordered if rule-sets are designed in positive logic), the time required for filtering grows in different orders depending on the algorithm and data structure used [7]. Conversely, performant algorithms, may require great memory occupation or dedicated hardware, which is another important parameter to take into account.
4. **Fault tolerance**. Firewalls inherently introduce a single point of failure in the network schema. Thus, a failure in the firewall results in temporary isolation of the protected network segments during reparation. Failures can arise due to hardware-related problems, such as problems in the power supply, bus, memory errors, etc. and software-related problems such as bugs. This can be overcome with redundancy and health check monitor techniques. The idea consists of having several firewall replicas: one that filters flows (primary replica), and others that (backup replicas) are ready to recover the services as soon as failure arises (See Fig. 1).

However, system-level redundancy is insufficient for *Stateful Firewalls*. Stateful firewalls extend the firewall capabilities to allow system administrators define state-based flow filtering. The stateful capabilities are enabled by means of the connection tracking system (CTS) [8]. The CTS performs a correctness check upon the protocols that it gateways. This is implemented through a finite state automaton for each supported protocol that determines what protocol transitions are valid. The CTS stores several aspects of the evolution of a flow in a set of variables that compose a *state*. This information can be used to deny packets that trigger invalid state transitions. Thus, the system administrator can use the states to define more intelligent and finer filter policies that provide higher level of security.

Let's assume the following example to clarify the fault-tolerance problem in stateful firewall environments: the primary firewall replica fails while there is an established TCP connection. Then, one of the backup replicas is selected to become primary and recover the filtering. However, since the new primary replica has not seen any packets for that existing connection, the CTS of the new primary firewall replica considers that TCP *PSH* packets of non-existing connections triggers an invalid state transition. With the appropriate stateful rule-set, this TCP connection will not be recovered since this packet triggers an invalid state transition (the first packet seen does not belong to any known established connection by the new primary firewall replica cannot be a TCP *PSH* packet). Thus, the packets are denied and the connection has to be re-established

[4]. Therefore, the redundant solution requires a replication protocol to guarantee that the flow states are known by all replica firewalls.

In this work, we specifically focus on solving the fault-tolerance problem. We extend the FT-FW (Fault Tolerant FireWall) [10], a reactive fault-tolerant solution for stateful firewalls, from the multiprimary setup perspective in which several replica firewalls can share workload. This solution guarantees transparency, simplicity, protocol independency, failure-detection independency and low cost. We extend our existing work to fulfil the scalability requirements of a multiprimary setting.

The main idea of our proposal is an event-driven model to reliably propagate states among replica firewalls in order to enable fault-tolerant stateful firewalls. The key concepts of FT-FW are the state proxy and the reliable replication protocol. The state proxy is a process that runs on every replica firewall and waits for events of state changes. This process propagates state changes between replicas and keeps a cache with current connection states. State propagation is done by means of the proposed reliable multicast IP protocol that resolves the replication.

The paper is organized as follows: in Section 3 we formalize the system model. In Section 4 we detail the architecture of FT-FW. The state proxy design is detailed in Section 4.1. The proposed replication protocol is described in Section 4.2. We focus on the specific multiprimary support in Section 5. Then we evaluate our solution proposed in Section 6 and detail the related work in Section 2. We conclude with the conclusions and future works in Section 7.

2 Related Work

Many generic architectures have been proposed to achieve fault-tolerance of network equipments with a specific focus on web servers and TCP connections [11] [12] [13] [14]. In these works, the authors cover scenarios where the complete state history has to be sent to the backup replicas to successfully recover the connections. Most of them are limited to 10/100 Mbit networks. These solutions can also be used to implement fault-tolerant stateful firewalls; however, they do not exploit the firewall semantics detailed in the system model (specifically definition 11). A state replication based on extra hardware has been also proposed [15]. Specifically, the authors use Remote Direct Memory Access (RDMA) mechanisms [15] to transfer states. This solution implies an extra cost and the use of a technology that may result intrusive and out of the scope of high performance computing clusters.

To the best of our knowledge, the only similar research in the domain of firewalls that we have found is [16]. This work is targeted to provide a fault-tolerant architecture for stateless firewalls with hash-based load-balancing support. We have used this idea in Sec. 5 to enable workload sharing without the need of a load balancing director.

[4] In our current work, we provide a detailed scenario in the website of the FT-FW implementation [9].

With regard to replication protocols suitable for CBSF, we have found TIPC [17] is a generic protocol designed for use in clustered computer environments, allowing designers to create applications that can communicate quickly and reliably with other applications regardless of their location within the cluster. TIPC is highly configurable and covers different cluster-based setups. This protocol is suitable for the scenario described in this work. The generic nature of TIPC makes it hard for it to fulfil the policies 1, 2 and 3.

In [18], the authors of this work propose preliminary design ideas and a set of problematic scenarios to define an architecture to ensure the availability of stateful firewalls. The authors of this work detail the FT-FW architecture from the Primary-Backup perspective in [10].

In the industry field, there are several proprietary commercial solutions such as CheckPoint Firewall-1, StoneGate and Cisco PIX that offer a highly available stateful firewall for their products. However, as far as we know, there is only documentation on how to install and configure the stateful failover. In the OpenSource world, the OpenBSD project provides a fault-tolerant solution for their stateful firewall [19]. The solution is embedded into the firewall code and the replication protocol is based on unreliable Multicast IP and it also has support for multiprimary setups. The project lacks of internal design documentation apart from the source code. Other existing projects such as Linux-HA [20] only focus on system-level fault-tolerance so it does not cover the problem discussed in our work.

3 Definitions and Notation

The formalization of the stateful firewall model is out of the scope of this work as other works have already proposed a model [21]. Nevertheless, we formalize the definitions extracted from the fault-tolerant stateful firewall semantics that are useful for the aim of this work:

Definition 1. Fault-tolerant stateful firewall cluster: it is a set of stateful replica firewalls $fw = \{fw_1, ..., fw_n\}$ where $n \geq 2$ (See Fig. 1). The number of replica firewalls n that compose the cluster depends on the availability requirements of the protected network segments and their services, the cost of adding a replica firewall, and the workload that the firewall cluster has to support. We also assume that failures are independent between them so that adding new replica firewalls improve availability. The set of replica firewalls fw are connected through a dedicated link and they are deployed in the local area network. We may use more than one dedicated link for redundancy purposes. Thus, if one dedicated link fails, we can failover to another.

Definition 2. Cluster rule-set: Every replica firewall has the same rule-set.

Definition 3. Flow filtering: A stateful firewall fw_x filters a set of flows $F_x = \{F_1, F_2, ..., F_n\}$.

Definition 4. Multiprimary cluster: We assume that one or more firewall replicas deploy the filtering at the same time, the so-called *primary replicas*, while others act as *backup replicas.*

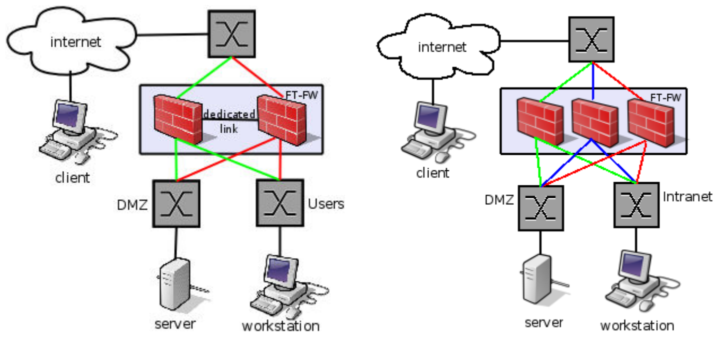

Fig. 1. Stateful firewall cluster of order 2 and order 3 respectively

Definition 5. Failure detection: We assume a failure detection manager, eg. an implementation of VRRP [22], that detects failures by means of heartbeat tokens. Basically, the replicas send a heartbeat token to each other every t seconds, if one of the replicas stops sending the heartbeat token, it is supposed to be in failure. This failure detection mechanism is complemented with several multilayer checkings such as link status detection and checksumming. This manager is also responsible of selecting which replica runs as primary and which one acts as backup. Also, we assume that the manager runs on every firewall replica belonging the cluster.

Definition 6: Flow durability (FD): The FD is the probability that a flow has to survive failures. If FD is 1 the replica firewall can recover all the existing flows. In this work, we introduce a trade-off between the FD and the performance requirements of cluster-based stateful firewalls.

Definition 7. Flow state: Every flow F_i in F is in a state S_k in an instant of time t.

Definition 8. State determinism: The flow states are a finite set of deterministic states $s = \{S_1, S_2, ..., S_n\}$.

Definition 9. Maximum state lifetime: Every state S_k has a maximum lifetime T_k. If the state S_k reaches the maximum lifetime T_k, we consider that the flow F_j is not behaving as expected, eg. one of the peers has shutdown due to a power failure without closing the flow appropriately.

Definition 10. State variables: Every state S_k is composed of a finite sets of variables $S_k = \{v_1, v_2, ..., v_j\}$. The change of the value of a certain variable v_a may trigger a state change $S_k \rightarrow S_{k+1}$.

Definition 11. State history: The backup replica does not have to store the complete state history $S_1 \rightarrow S_2 \rightarrow ... \rightarrow S_k$ to reach the consistent state S_k. Thus, the backup only has to know the last state S_k to recover the flow F_i.

Definition 12. State classification: The set of states s can be classified in two subsets: transitional and stable states. These subsets are useful to notice if the effort required to replicate one state change is worthwhile or not:

– *Transitional states* (TS) are those that are likely to be superseded by another state change in short time. Thus, TS have a very short lifetime.
– *Stable States* (SS) are long standing states (the opposite of TS).

We have formalized this state classification as the function of the probability (P) of the event of a state change (X). Let t be the current state age. Let T_k be the maximum lifetime of a certain state. For the flow F_j the current state S_k, we define the probability P_x that a TS can be superseded by another state change can be expressed as:

$$P_x(t, S_k) = \begin{cases} 1 - \delta(t, S_k) & if\ (0 \leq t < T_k) \\ 0 & if\ (t \geq T_k) \end{cases}$$

And the probability P_y that a SS can be superseded by a state change can be expressed as:

$$P_y(t, S_k) = 1 - P_x(t, S_k)$$

This formalization is a representation of the probability that a state can be replaced by another state as time goes by. Both definitions depend on the $\delta(t, S_k)$ function that determines how the probability of a state change S_k increases, e.g. linearly, exponential, etc. The states can behave as SS or TS depending on their nature, eg. initial TCP handshake and closure packets (*SYN, SYN-ACK* and *FIN, FIN-ACK, ACK* respectively) trigger TS and TCP *ACK* after *SYN-ACK* triggers TCP Established which usually behaves as SS. Network latency is another important factor because if latency is high, all the states tend to behave as SS. In practise, we can define a simple $\delta(t, S_k)$ that depends on the acceptable network latency l:

$$\delta_x(t, S_k) = \begin{cases} 1 & if\ \ t > (2 * l) \\ 0 & if\ \ t \leq (2 * l) \end{cases}$$

The acceptable network latency l depends on the communication technology, eg. on a wired line the acceptable latency is 100 ms and in satellite links 250 ms.

For the aim of this work, we focus on ensuring the durability of SS as they have a more significant impact on the probability that a flow can survive failures. This means that our main concern is to ensure that long standing flows can survive failures because the interruption of these flows lead to several problems such as:

1. Extra monetary cost for an organization, eg. if the VoIP communications are disrupted, the users would have to be re-called with the resulting extra cost.
2. Multimedia streaming applications breakage, eg. Internet video and radio broadcasting disruptions.
3. Remote control utility breakage, eg. SSH connections closure.
4. The interruption of a big data transfer between two peers, eg. peer to peer bulk downloads.

Nevertheless, the high durability of TS is also desired; however, they are less important than SS since their influence on the FD is smaller.

4 FT-FW Architecture

The FT-FW architecture is composed of two blocks: the state proxy and the efficient and scalable replication protocol.

4.1 State Proxy

From the software perspective, each replica firewall is composed of two parts:

1. The connection tracking system (CTS): the system that tracks the state evolution of the connections. This software block is part of a stateful firewall, and the packet filter uses this state information to filter traffic [8].
2. The *state proxy* (SP): the application that reliably propagates state changes among replica firewalls [23].

In order to communicate both parts, we propose an event-driven architecture (EDA) which provides a natural way to propagate changes: every state change triggers an event that contains a tuple composed of $\{Address_{SRC}$, $Address_{DST}$, $Port_{SRC}$, $Port_{DST}$, $Protocol\}$, that uniquely identifies a flow, together with the set of variables that compose a state $S_k = \{v_1, v_2, ..., v_n\}$. Thus, the CTS sends events in response to state changes. These events are handled by the SP which decides what to do with them. We have classified events into three types [18]: *new*, which details a flow that just started; *update*, which tells about an update in an opened flow and *destroy*, which notifies the closure of an existing flow.

The EDA facilities modularization and reduces dependencies since the CTS and the SP are loosely coupled. Moreover, its asynchronous nature suits well for the performance requirements of stateful firewalls.

We have modified the CTS to implement a framework to subscribe to state change events, dump states and inject them so that the SP can interact with the CTS. The number of changes required to introduce this framework in the CTS is minimal. This framework makes the FT-FW architecture independent of the CTS implementation since we clearly delimit the CTS and the SP functionalities. Also, the FT-FW solution allows the system architect to add support for fault tolerance in a plug-and-play fashion, ie. the system architect only has to launch the SP in runtime and add new replica firewalls to enable FT-FW. The CTS framework offers three methods to the SP:

1. *Dumping*: it obtains the complete CTS state table, including generic and specific states. This method is used to perform a full resynchronization between the SP and the CTS.
2. *Injection*: it inserts a set of states, this method is invoked during the connection failover.
3. *Subscription*: it subscribes the SP to state-change notifications through events.

The SP listens to events of state change, maintains a cache with current states, and sends state-change notifications to other replicas. We assume that every replica firewall that is part of the cluster runs a SP. Every SP has two caches:

– The *internal cache* which holds local states, ie. those states that belong to flows that this replica is filtering. These states can be a subset of states $subset(s)$ of the set of states s held in the CTS. This is particularly useful if the system architect does not want to guarantee the FD of certain flows whose nature is unreliable, eg. the UDP name resolution flows (UDP DNS) that are usually reissued in short if there is no reply from the DNS server. Thus, we assume that the CTS provides an event filtering facility to ignore certain flows whose state the SP does not store in the internal cache.
– The *external cache* which hold foreign states, ie. those states that belong to connections that are not being filtered by this replica. If the firewall cluster is composed of n replicas, the number of external caches is $n - 1$ at maximum. Thus, there is an external cache for every firewall replica in the cluster so that, when a failure arises in one of the firewall replicas fw_y, one of the backups fw_x is selected to inject the flow states stored in its external cache fw_y.

We represent the FT-FW architecture for three replica firewalls and the interaction between the blocks in Fig. 2. Note that, in this particular case, the number of external caches is two so that every replica firewall can recover the filtering of the other two replicas at any moment.

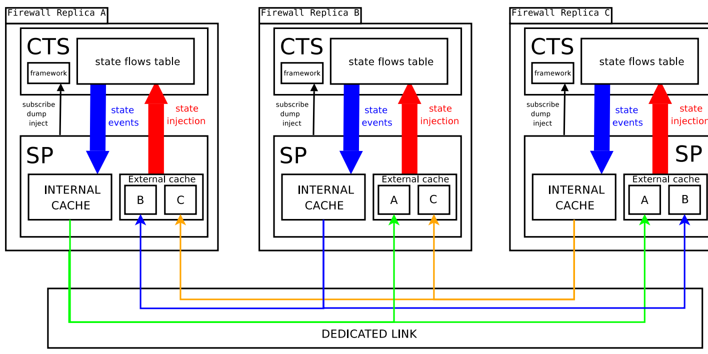

Fig. 2. FT-FW Architecture of order 3

At startup, the SP invokes the dumping method to fully resynchronize its internal cache with the CTS, and subscribes to state change events to keep the internal cache up-to-date. The flows are mapped into a *state objects* which are stored in the internal cache. We also assume that the events that the CTS generates are mapped into temporary state objects that are used to update the internal cache.

Definition 13. State object: We assume that every flow F_j is mapped into an *state object* (SO). This SO has an attribute *lastseq_seen* to store the last sequence number m of the message sent that contained the state change $S_{k-1} \rightarrow S_k$. This sequence number is updated when *send_msg()* is invoked.

The purpose of this sequence number attribute is to perform an efficient state replication under message omission and congestion situations as we detail in the replication protocol section.

The operation of the SP consists of the following: A packet p that is part of an existing flow F_j may trigger a state change $S_{k-1} \rightarrow S_k$ when the primary replica firewall succesfully finds a match in the rule-set for p. If such state change occurs, it is notified through an event delivered to the SP. The SP updates its internal cache and propagates the state change to other replicas via the dedicated link (See Algorithm. 1 for the implementation of the internal cache routine). Thus, the backup firewall SPs handle the state change received and insert it in their external cache (See Algorithm. 2 for the implementation of the external cache routine).

```
internal ← create_cache();
dump_states_from_CTS(internal);
subscribe_to_CTS_events();
for ever do
    object ← read_event_from_CTS();
    switch event_type(object) do
        case new
            cache_add(internal, object);
        end
        case update
            cache_update(internal, object);
        end
        case destroy
            cache_del(internal, object);
        end
    end
    send_msg(object);
end
```

Algorithm 1: Internal cache routine

The function *send_msg*() converts the object which represents the state change event into network message format and sends it to the other replicas. The function *recv_msg*() receives and converts the network message format into a state object. The implementation of these functions is discussed in the replication protocol.

4.2 Replication Protocol

In this work, we propose an asynchronous replication protocol to replicate state changes between replica firewall. This protocol trades off with the FD (definition 6) and performance. The FT-FW protocol also handles link congestions and mes-

```
external ← create_cache();
request_resync(external);
for ever do
    object ← read_msg();
    switch event_type(object) do
        case new
            cache_add(external, object);
        end
        case update
            cache_update(external, object);
        end
        case destroy
            cache_del(external, object);
        end
    end
end
```
Algorithm 2: External cache routine

sage omission situations efficiently by exploiting the stateful firewall semantics, specifically *definition 11*.

Definition 14. Message omission handling: Given two messages with sequence number m and $m + k$ that contains state changes $S_{k-2} \rightarrow S_{k-1}$ and $S_{k-1} \rightarrow S_k$ respectively. If both messages are omitted, only the state change $S_{k-1} \rightarrow S_k$ is retransmitted since, due to *definition 11*, the old state changes, such as $S_{k-2} \rightarrow S_{k+1}$ does not improve the FD.

Replication has been studied in many areas, especially in distributed systems for fault-tolerance purposes and in databases [24] [25]. These replication protocols (RP) may vary from synchronous to asynchronous behaviours:

1. *Synchronous* (also known as eager replication): These RPs are implemented through transactions that guarantee a high degree of consistency (in the context of this work, this means a FD close to 1). However, they would roughly reduce performance in the cluster-based stateful firewall environment. With a synchronous solution, the packets that trigger state changes must wait until all backup replicas have successfully updated their state synchronously. This approach would introduce an unaffordable latency in the traffic delivery. The adoption of this approach would particularly harm real-time traffic and the bandwidth throughput.
2. *Asynchronous* (also known as lazy replication): This approach speeds up the processing in return of it reduces the level of consistency between the replicas and increasing the complexity. From the database point of view, a high degree of data consistency is desired so this approach usually makes asynchronous solutions unsuitable. However, in the context of stateful firewalls, the asynchronous replication ensures efficient communication which helps to avoid quality of service degradation.

Therefore, we have selected an asynchronous solution which allows the packet to leave the primary firewall before the state has been replicated to other backup replicas. We propose an efficient and reliable replication protocol for cluster-based stateful firewalls (CBSF) based on Multicast IP. Our protocol uses sequence tracking mechanisms to guarantee that states propagate reliably. Although message omissions are unlikely in the local area, communication reliability is a desired property of fault-tolerant systems.

In our protocol, we define three kinds of messages that can be exchanged between replicas, two of them are control messages (Ack and Nack) and one that contains state changes:

- *Positive Acknowledgment* (Ack) is used to explicitly confirm that a range of messages were correctly received by the backup replica firewall.
- *Negative Acknowledgment* (Nack) explicitely requests the retransmission of a range of messages that were not delivered.
- *State Data* contains the state change $S_{k-1} \rightarrow S_k$ for a given flow F_j. This message contains the subset of variables $v = \{v_1, ..., v_n\}$ that has changed.

Our replication protocol is based on an incremental sequence number algorithm and it is composed of two parts: the sender and the receiver. The sender and the receiver are implemented through *send_msg()* and *recv_msg()* respectively (See Algorithm.3 and Algorithm. 4). Basically, the sender transmits state changes and control messages and the receiver waits for control messages, which request explicit retransmission and confirm correct reception.

We formalize the behaviour of the replication protocol with the following policies:

Policy 1. Sender Policy: The sender does not wait for acknowledgments to send new data. Thus, its behaviour is asynchronous since it never stops sending state changes.

Policy 2: Receiver policy: The receiver always delivers the messages received even if they are out of sequence. This policy is extracted from the *definition 11.*

Policy 3: Receiver acknowledgment policy: The receiver schedules an acknowledgment when we receive $WINDOW\_SIZE$ messages correctly, and negative acknowledges the range of those messages that were not delivered appropriately. The best value of $WINDOW\_SIZE$ is left for future works due to space restrictions.

5 Multiprimary support

The FT-FW architecture supports several workloads sharing multi-primary setups in which several replica firewalls act as primary. Thus, more than one replica firewall can filter traffic at the same time. This is particularly important to ensure that the solution proposed scales up well. Specifically, our solution covers two approaches: the symmetric and the asymmetric path workload sharing.

```
switch typeof(parameters) do
    case Ack
        msg ← build_ack_msg(from, to);
    end
    case Nack
        msg ← build_nack_msg(from, to);
    end
    case Data
        object.lastseq_seen = seq;
        if is_enqueued(retransmission_queue, object) then
            queue_del(retransmission_queue, object);
            queue_add(retransmission_queue, object);
        else
            queue_add(retransmission_queue, object);
        end
        msg ← build_data_msg(seq, object);
    end
    send(msg);
    seq ← seq + 1;
end
```

Algorithm 3: Implementation of *send_msg*()

Symmetric path: in this approach, the same replica firewall always filters the original and reply packets. Therefore, we apply per-flow workload sharing. Thus, the replica firewalls can act as primary for a subset F_1 of flows and as backup another subset of flows F_2 at the same, being $F_1 \; U \; F_2$ the complete set of flows that both firewalls are filtering.

For the symmetric path approach. We consider two possible setups depending on the load balancing policy, they are:

- *Static.* The system administrator or the DHCP server configures the clients to use different firewalls as gateway, ie. the client A is configured to use the gateway G_1 and the client B uses the gateway G_2. And so, if the gateway G_2 fails, the gateway G_1 takes over B's connections. Thus, the same firewall filters traffic for the same set of clients (statically grouped) until failure.
- *Dynamic.* Flows are distributed between replica firewalls by means of hash-based load balancing similar to what is described in [16]. The tuple $t = \{Address_{SRC}, Address_{DST}, Port_{SRC}, Port_{DST}\}$ which identifies a flow F_j is used to determine which replica filters each flow. Basically, the tuple t is hashed and the modulo of the result by the number of replicas tells which replica has to filter the flow, eg. given two replicas fw_0 and fw_1, if $h(t) \; mod$ 2 returns 0 then the replica firewall fw_0 filters the flow. For this solution we assume that all replica firewalls use a multicast MAC address and the same IP configuration so that they all receive the same packets. This approach does not require any load balancing director.

```
msg ← recv();
n ← msg.sender_node;
if after(msg.seq, lastseq_seen[n] + 1) then
    confirmed ← WINDOW_SIZE - window[n];
    send_msg(Ack, n, lastseq_seen[n] - confirmed, lastseq_seen[n]);
    send_msg(Nack, n, lastseq_seen[n] + 1, msg.seq);
    window[n] ← WINDOW_SIZE;
else
    window[n] ← window[n] - 1;
end
if window[n] = 0 then
    window[n] ← WINDOW_SIZE;
    from ← msg.seq - WINDOW_SIZE;
    send_msg(Ack, n, from, msg.seq);
end
if msg_type(msg) = Ack then
    foreach object i in the retransmission_queue[n] do
        if between(msg.from, seq(i), msg.to) then
            queue_del(object);
        end
    end
end
if msg_type(msg) = Nack then
    foreach object i in the retransmission_queue[n] do
        if between(msg.from, seq(i), msg.to) then
            send_msg(object);
        end
    end
end
lastseq_seen[n] ← msg.seq;
deliver(msg)
```

Algorithm 4: Implementation of *recv_msg*()

The external cache policy in symmetric path is *write back* (WB), ie. the states are only injected to the CTS in case of failure.

Asymmetric path: in this setup, any replica firewall may filter a packet that belongs to a flow. Therefore, we apply per-packet workload sharing. In this case, we assum that the original and reply packets may be filtered by different replica firewalls. Again, we consider two possible setups depending on the workload sharing policy, they are:

- *Static.* The system administrator has configured firewall fw_n as default route for original packets and fw_{n+1} as default route for reply packets.
- *Dynamic.* The routes for the original and reply packets may dynamically change based on shortest path first routing policies, as it happens in OSPF

[26] setups. In this case, the firewall dynamically configures the routing table depending on several parameters such as the link latency.

The external cache policy behaviour is *write through* (WT) since SP injects the states to the CTS as they arrive. Of course, asymmetric path setups incur an extra penalty in terms of CPU consumption that has to be evaluated.

5.1 Flow Recovery

As said, the architecture described in this work is not dependent of the failure-detection schema. So, we assume a failure-detection software, e.g. an implementation of VRRP.

If the primary firewall fails, the failure-detection software selects the candidate-to-become-primary replica firewall among all the backup replicas that will take-over the flows. At the failover stage, the recovery process depends on the load balancing setup:

– Symmetric path load balancing: the selected replica firewall invokes the injection method that puts the set of states stored in the external cache into the CTS. Later on, the SP clears its internal cache and issues a dump to obtain the new states available in the CTS.
– Asymmetric path load balancing: since the external cache policy is WT, the states are already in the CTS.

If a backup replica firewall fails and, later on, comes back to life again (typical scenario of short-time power-cut and reboot or a maintenance stop), the backup replica that just restarted sends a full resynchronization request. If there is more than one backup, to reduce the workload of the primary replica, that backup may request the full state table to another backup replica. Moreover, if this backup was a former primary replica that has come back to life, we prevent any process of take-over attempt by such replica until it is fully resynchronized.

6 Evaluation

To evaluate FT-FW, we have used our implementation of the state proxy daemon for stateful firewalls [27]. This software is a userspace program written in C that runs on Linux. We did not use any optimization in the compilation. In our previous work [10], we have already evaluated the recovery time of the connections in the Primary-Backup scenario, these results are similar to those obtained in the multiprimary setup, for that reason, and due to space restrictions we do not provide them in this section.

The testbed environment is composed of AMD Opteron dual core 2.2GHz hosts connected to a 1 GEthernet network. The schema is composed of four hosts: host A and B that act as workstations and FW1 and FW2 that are the firewalls. We have adopted a Multiprimary configuration with workload sharing of order two for simplicity. Thus, both FW1 and FW2 acts as primary for each other at the same time. In order to evaluate the solution, we reproduce a hostile scenario in which one of the hosts generates lots of short connections, thus generating loads of state change messages. Specifically, the host A requests HTML files of 4 KBytes to host B that runs a web server. We created up to 2500 GET HTTP requests per second (maximum connection rate reached with the testbed used). For the test case, we have used the Apache webserver and a simple HTTP client for intensive traffic generation.

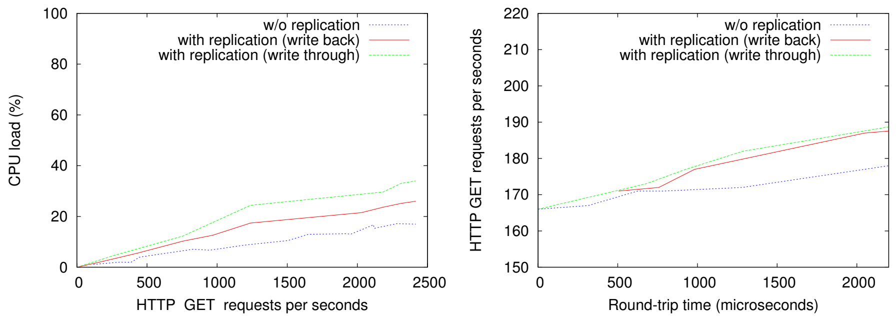

Fig. 3. CPU consumption and Round-trip time (from left to right)

6.1 CPU overhead

We have measured CPU consumption in FW1 and FW2 with and without full state replication. The tool *cyclesoak* [28] has been used to obtain accurate CPU consumption measurements. The HTTP connections have 6 states, thus the amount of state changes is *6 * total number of requests.* The results obtained in the experimentation have been expressed in a graphic. In both firewalls, the maximum CPU load is 24% and 36% for WB and WT external cache policy respectively. This means 9% and 17% more than without replication. Not surprisingly, the full replication of short connection is costly due to the amount of states propagated. Anyhow, the CPU consumption observed is affordable for CBSFs deployed on off-the-shelf equipments since they come with several low cost processors (SMP and hyperthreading). Thus, we can conclude that FT-FW guarantees the connection recovery at the cost of requiring extra CPU power.

6.2 Round Trip

In order to obtain the delay that FT-FW introduces in client responses, we have measured the round-trip time of an ICMP echo request/reply (ping pong time) from host A to B with and without replication enabled. The results has been expressed in Fig. 3. As we can observe, the increment in the round trip time is around 8 microseconds so that we can say that the delay introduced in clients' responses is negligible.

7 Conclusion and Future Work

In this work we have revisited the FT-FW (Fault Tolerant FireWall) solution from the multiprimary perspective. The solution introduced negligible extra network latency in the packet handling at the cost of relaxing the replication. The architecture follows an event-driven model that guarantees simplicity, transparency, fast client responses and quick recovery. No extra hardware is required. The solution proposed is not dependent of the failure detection schema nor the layer 3 and 4 protocols that the firewalls filter. The FT-FW replication protocol exploits the cluster-based stateful firewall semantics to implement an efficient replication protocol. Moreover, we have proved in the evaluation that the solution requires affordable CPU resources to enable state replication.

As future work, we are dealing with several improvements to reduce CPU consumption without harming FD in environments with limited resources such as mobile and embedded systems. Specifically, we plan to use our state-classification model to avoid the replication of TS since they barely improve FD but they increase resource consumption due to the state replication.

References

1. A. Wool, "A quantitative study of firewall configuration errors," *IEEE Computer*, vol. 37, no. 6, pp. 62–67, 2004.
2. A. Mayer, A. Wool, and E. Ziskind, "Offline Firewall Analysis," *International Journal of Computer Security*, vol. 5, no. 3, pp. 125–144, 2005.
3. E. Al-Shaer and H. Hamed, "Taxonomy of Conflicts in Network Security Policies," *IEEE Communications Magazine*, vol. 44, no. 3, 2006.
4. E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan, "Conflict Classification and Analysis of Distributed Firewall Policies," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 23, no. 10, 2005.
5. Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: A novel firewall management toolkit," *ACM Transactions on Computer Systems*, vol. 22, no. 4, pp. 381–420, Nov. 2004.
6. S. Pozo, R. Ceballos, and R. M. Gasca, "CSP-Based Firewall Rule Set Diagnosis using Security Policies," *2nd International Conference on Availability, Reliability and Security*, 2007.
7. D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238–275, 2005.
8. P. Neira, "Netfilter's Connection Tracking System," *In :LOGIN;, The USENIX magazine*, vol. 32, no. 3, pp. 34–39, 2006.
9. ——, "Conntrack-tools: Test Case," 2007. [Online]. Available: http://people.netfilter.org/pablo/conntrack-tools/testcase.html
10. P. Neira, R. M. Gasca, and L. Lefevre, "FT-FW: Efficient Connection Failover in Cluster-based Stateful Firewalls," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'08)*, Feb. 2008, pp. 573–580.
11. R. Zhang, T. Adelzaher, and J. Stankovic, "Efficient TCP Connection Failover in Web Server Cluster," in *IEEE INFOCOM 2004*, Mar. 2004.

12. M. Marwah, S. Mishra, and C. Fetzer, "TCP server fault tolerance using connection migration to a backup server," *In Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pp. 373–382, Jun. 2003.
13. N. Aghdaie and Y. Tamir, "Client-Transparent Fault-Tolerant Web Service," in *20th IEEE International Performance, Computing, and Communication conference*, 2001, pp. 209–216.
14. N. Ayari, D. Barbaron, L. Lefevre, and P. Primet, "T2CP-AR: A system for Transparent TCP Active Replication," in *AINA '07: Proceedings of the 21st International Conference on Advanced Networking and Applications*, 2007, pp. 648–655.
15. F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode, "Recovering Internet Service Sessions from Operating System Failures," in *IEEE Internet Computing*, Apr. 2005.
16. R. M. Y. Chen, "Highly-Available Firewall Service using Virtual Redirectors," University of the Witwatersrand, Johannesburg, Tech. Rep., 1999.
17. J. Maloy, "TIPC: Transparent Inter Protocol Communication protocol," May 2006.
18. P. Neira, L. Lefevre, and R. M. Gasca, "High Availability support for the design of stateful networking equipments," in *Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06)*, Apr. 2006.
19. R. McBride, "Pfsync: Firewall Failover with pfsync and CARP." [Online]. Available: http://www.countersiege.com/doc/pfsync-carp/
20. A. Robertson, "Linux HA project." [Online]. Available: http://www.linux-ha.org
21. M. Gouda and A. Liu, "A model of stateful firewalls and its properties," *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 128–137, Jun. 2005.
22. R. Hinden, "RFC 3768: Virtual Router Redundancy Protocol (VRRP)," Apr. 2004.
23. X. Zhang, M. A. Hiltunen, K. Marzullo, and R. D. Schlichting, "Customizable Service State Durability for Service Oriented Architectures," in *IEEE Proceedings of EDCC-6: European Dependable Computing Conference*, Oct. 2006, pp. 119–128.
24. M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding Replication in Databases and Distributed Systems," *International Conference on Distributed Computing Systems*, pp. 464–474, 2000.
25. R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso, "How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead," *International Conference on Reliable Distributed Systems*, p. 24, 2001.
26. J. Moy, "RFC 1247 - OSPF Version 2," Jul. 1991.
27. P. Neira, "conntrackd: The netfilter's connection tracking userspace daemon." [Online]. Available: http://people.netfilter.org/pablo/
28. A. Morton, "cyclesoack: a tool to accurately measure CPU consumption on Linux systems." [Online]. Available: http://www.zip.com.au/ akpm/linux/