Abstract—Stateful firewalls are security solutions widely deployed in the Internet. These devices filter network traffic and keep track of the state of connections in order to make the deployment of several attacks, such as TCP resets, difficult. However, firewalls are critical equipments in the network schema since they introduce a single point of failure. Therefore, a failure may isolate networks, users and interrupt established connections. Current fault tolerant solutions mask failures by means of replication techniques based on physical redundancy and state propagation. However, these solutions do not suit well for stateful firewall scenarios since they reduce bandwidth throughput roughly, they require costful extra hardware or are stuck to wasteful and inflexible single primary-backup settings. In this work we detail FT-FW (Fault Tolerant FireWall), a software-based transparent connection failover mechanism for stateful firewalls. Our solution has a negligible impact in terms of performance, as well as the fact that quick recovery from failures and fast responses to clients are guaranteed. The architecture is suitable for low cost off-the-shelf systems and no extra hardware is required.

I. INTRODUCTION

Firewalls are perimeter security devices widely deployed in the Internet. These equipments allow system administrators to define filtering policies that determine the allowed traffic. These filtering policies are defined by means of rule-sets, containing each rule a set of descriptors that match packet fields and the action to be issued, such as accept or deny.

However, this approach has been demonstrated to be insufficient against several attacks such as TCP resets [1]. For that reason, a new generation of firewalls, the so-called stateful firewalls, implement connection tracking. This technique guarantees that connections evolve in a standard-compliant manner, making it more difficult to deploy these attacks. The connection tracking stores the connection evolution in a variable known as state that represents the current status of the connection, and determines which state transitions are valid and which are not. For the sake of simplicity, we classify connection states into two types:

1) New: the client has sent the first packet to initiate a connection. So, we say that a connection is in new state if this is the first packet seen by the firewall.

2) Established: the firewall has seen packets in both directions. Therefore, the client and the server have already exchanged at least two packets to reach this state.

An example of a simple stateful ruleset (assuming default policy is deny) can be:

\[
\text{if tcp and state=established, accept} \\
\text{if tcp syn dport=22 and state=new, accept}
\]

Basically, the first rule allows any kind of established connections and the second allows packets with the flag syn set from clients to SSH servers. Thus, the first packet that initiates a connection to an SSH server matches the second rule and consequently the next packets leaving the server and the client match the first rule since they belong to an already established connection. Note that the concept of established is not related to the TCP established state. Therefore, UDP flows also have states for the firewall even if their nature is stateless. In short, the state variable allows system administrators to define more intuitive and intelligent filtering policies with just a few rules while defeating attacks.

On the other hand, firewalls inherently introduce a single point of failure since all the traffic has to cross the firewall to enter and leave the network that it protects. Therefore, a failure in the firewall results in temporary isolation during the repairing time. This can be overcome with physical redundancy and health check monitor techniques. The idea consists of the use of a cluster composed of two or more firewall devices: some of them filter connections (primary replicas), and some others are ready to recover the service as soon as failures arise, which are also known as backup replicas (Fig. 1). Thus, if the primary replica fails, one of the backup replicas is selected to recover the established connections.

However, this approach is insufficient for stateful firewalls. Basically, the problem resides in the state variable. Since the backup replica does not know the state variable, the connection may not be appropriately recovered, leading to disruptions. Let’s assume the following scenario based on the example ruleset and the firewall cluster-based setting: a client establishes a SSH connection with a server that is protected by the stateful firewall. Therefore, the current state of that
connection is established. Then, the client sends a packet that successfully reaches the SSH server when a failure arises, which makes a backup replica take over the filtering. Then the SSH server sends a reply packet to the client but, unfortunately, it will match neither the first rule, since that packet does not belong to any known established connection by the former backup, nor the second since the destination port is a client port > 1024. Therefore, after several attempts by the client and the server to continue the communication, the connection will timeout. Thus, the connection has to be re-established. In our current work, a more detailed and realistic ruleset is described in [2].

In this work we focus on providing a solution to recover connections under failure situations for cluster-based stateful firewalls (CBSF). Our solution satisfies the following requirements:

1) Transparency. It must guarantee negligible delay in client responses and quick recovery from failures.
2) Simplicity. We reuse and extend existing software-based, high availability solutions. Moreover, the client does not require any modification. Therefore, this is a client transparent solution. The firewall must also require minimal and non-intrusive modifications.
3) Fast responses to clients. The solution proposed must have a negligible impact in terms of performance, i.e. clients must not notice any bandwidth throughput drop, and it must be suitable for 1 GEthernet network setup.
4) Low cost. The solution must be suitable for off-the-shelf equipments. No hardware extensions must be required.

The main idea of our proposal is an event-driven model to reliably propagate states among replica firewalls in order to enable fault tolerant CBSFs. 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 firewall replica and waits for events of state changes. This process propagates states changes between replicas and keeps a cache with current connection states. State propagation is done by means of the proposed reliable unicast that resolves the replication problem better than any existing reliable protocols.

The paper is organized as follows: in Section II we detail the architecture of FT-FW. The proposed replication protocol is described in Section III. Then we evaluate our solution proposed in Section IV and detail the related work in Section V. We conclude with the discussion in Section VI, and the conclusions and future works in Section VII.

II. ARCHITECTURE OF FT-FW

A. Overview

The firewall cluster is composed of two or more replica firewalls (Fig. 1) always coupled two by two. Such replicas are deployed in the local area network. We assume that, at least, one of the replicas filters traffic and they all have the same rule-set, so they deploy the same filtering policy. Each replica has a software subsystem, the so-called connection tracking system [3], which tracks connection evolution. The generic and specific state information is stored in a hash table for efficiency. We also assume that replicas are connected through a dedicated link that is used to propagate state changes.

In this work, the meaning of stateful depends on whether we refer to firewalls or protocols. We assume that, in the context of stateful firewalls, all protocols are stateful and flows can be, as we explained in the introduction, in two generic states: new and established, so that UDP flows also have states even if this is a stateless protocol. Of course, since TCP is a stateful protocol (that has its own states), the connection tracking can do closer inspection than UDP, and it can also internally maintain the current TCP state to check for valid TCP state transitions. Since the connection tracking must only inspect packets and does not deny traffic, we require a new state to explicitly drop invalid protocol sequences that we call invalid. An example of the use of this state in a rule can be:

\[
\text{if tcp and state=invalid, deny}
\]

Thus, any TCP packet that triggers an invalid protocol state transition is marked as invalid by the connection tracking so that it can be denied with the previous example rule. Of course, UDP flows never reach invalid states since UDP has no specific protocol states although it does have generic states (as described in the introduction). In this work we specifically focus on TCP connection filtering in the evaluation, but the proposed solution supports all other transport protocols both stateful and stateless.

B. State Proxy

We propose an event-driven model (EDM) that facilitates modularization and whose asynchronous nature meets well for the performance requirements of stateful firewalls. Moreover, the EDM is suitable for distributed systems and provides loose coupling so that the solution can be easily adapted to changes. The EDM provides a natural way to propagate changes: every state change triggers an event that contains a tuple composed of \(\{\text{Address}_{SRC}, \text{Address}_{DST}, \text{Port}_{SRC}, \text{Port}_{DST}\}\), that uniquely identifies a flow, together with the generic and
the specific states. Events are handled by the state proxy which
decides what to do with them.

The state proxy (SP) is one of the key concepts of this
architecture. Basically, the SP guarantees that states are prop-
agated reliably among replica firewalls [4]. Our SP is a daemon
that 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. Our implementation of the connection
tracking system (CTS) provides a framework to subscribe to
state change events, dump and inject states so that the SP can
interact with the CTS. This framework offers three methods:

1) Dumping: it obtains the complete CTS state table, in-
cluding 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 noti-
fications through events.

Every SP has two caches, one to hold local states, i.e. those
states that belong to flows that this replica is filtering, the
so-called internal cache, and another one for foreign states,
i.e. those states that belong to connections that are not being
filtered by this replica, the so-called external cache.

Initially, the SP invokes the dumping method to fully resyn-
chronize its internal cache with the CTS, and subscribes to
state change events to keep the internal cache up-to-date. State
changes are propagated via the dedicated link to other SPs that
keep foreign states in the external cache. Thus, the replication
consists of the following steps: a packet that belongs to a
flow triggers a state change when the primary firewall replica
performs the filtering and accepts it. Then, this state change
is notified through an event delivered to the SP. The SP then
updates its internal cache and propagates the state change to
other replicas via the dedicated link. This way, the backup
firewall SPs handle the state change received and insert it in
their external cache. We represent the FT-FW architecture in
Fig. 2.

C. Connection Failover

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 [5] [6].

If the primary replica firewall fails, the failure-detection
software selects the candidate-to-become-primary replica fire-
wall among all the backup replicas that will takeover the flows.
At the failover stage, 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 daemon clears its
internal cache and issues a dump to obtain the new states
available 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 maintainance 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 takeover
attempt by such replica until it is fully resynchronized.

III. FT-FW replication protocol

As said before, states need to be propagated reliably among
replicas since they are crucial to successfully recover flows
at the failover stage. For that task, we have to use a reliable
replication protocol that must improve the current state dur-
ability (CSD), which is the probability that the current state of
a flow has to survive failures [4]. Thus, the backup replicas
will be able to recover as much flows as possible during the
failover.

Initially, we could use any existing reliable transport pro-
tocol such as TCP to propagate state changes. However, TCP
can reduce CSD because of the latency introduced by in-
order delivery and congestion control mechanisms. The latency
problem is extracted from the following assumptions: in TCP,
a new bunch of data can be sent if the previous data sent was
acknowledged, and the size of the data sent cannot exceed the
receiver’s buffer size. Basically, this means that no new state
changes $S_{i+1}, S_{i+2}, ..., S_n$ for the flow $F_j$ can be sent
until the previous $S_i$ has been correctly received. Thus, the backups
hold a superseded state $S_i$ while the primary holds $S_{i+1}$ for
a flow $F_j$ during $l$ seconds which equals the latency that TCP
introduces. Since latency reduces CSD, the probability that the
flow $F_j$ can be appropriately recovered during the failover
decreases. Therefore, the replication protocol used must reduce
latency to improve CSD. Moreover, under message omission
situations, such as problems in the dedicated link, TCP ordered
data transfer also reduces CSD. Let’s assume the following
scenario: the primary firewall sends several state change
messages that are omitted due to a problem in a switch
that is part of the dedicated link. So, the TCP stack of the
backup notices message loss and requests retransmission via
acknowledgement, considering that the message with sequence
number $k$, which contains a state change $S_i$ for the flow
$F_j$, is omitted several times. Meanwhile, $F_j$ state is updated,
therefore $S_i \rightarrow S_{i+1}$. Once the message $k$ is successfully received, another message $m+n$ will follow later on, to update the state of $F_j$. However, CSD decreases since the delivery of $m+n$ is delayed until $m$ is successfully delivered. We have also studied other existing transport protocols such as RUDP [7], and SCTP [8] but they similarly reduce CSD. We have found that DCCP [9] has a similar concern: it removes long delays that TCP can cause. However, DCCP is targeted to streaming applications.

In this work, we propose an efficient and reliable replication protocol for cluster-based stateful firewalls (CBSF) based on UDP. However, since UDP does not provide reliable communication, we have to add sequence-tracking mechanisms to guarantee that states are propagated reliably. Our UDP-based replication protocol must improve CSD (by reducing latency) and efficiently handle message omission situations.

The replication protocol proposed is based on an incremental sequence number algorithm and it is composed of two parts: the sender and the receiver. Basically, the sender transmits state changes and control messages. The receiver waits for control messages, which request explicit retransmission and confirm correct reception. We assume that each flow $F_j$ is represented through an object, the so-called state object (SO), that stores the current state $S_j$ and the sequence number $m$ of the message sent that contained such state change $S_j$.

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 where correctly received by the backup replica firewall.
- **Negative Acknowledgment** (NACK) explicitly requests the retransmission of a range of messages that were not delivered.
- **State Data** that contains state changes.

The sender and the receiver use three shared structures to communicate with each other. The most important one is the backlog list (bk_list) which contains a list of SOs that are waiting to confirm the correct delivery of the last state change message sent. Thus, an SO is inserted in the bk_list once a state change happens and the corresponding state change message is sent, and removed when the message delivery is confirmed via ACK. We also introduce two queues, the data_queue and the ack_queue. Since the receiver part does not send messages, we define two queues that the receiver uses to tell to the sender what messages it must transmit. Thus, the data_queue contains state changes that need to be retransmitted (due to message omission) and the ack_queue contains ACK and NACK messages.

The main idea of the sender part is that it does not wait for acknowledgements to send new data, thus it never stops sending state changes. The state change messages are sent via the send_state() method that annotates the sequence number $m$ used in the message and inserts the SO in the bk_list. If the SO already exists in the list, it is removed and reinserted again at the end of the list to keep the bk_list ordered by sequence number so that search time is reduced. In short, the task of the sender consist on 1) waiting for state changes to be propagated and 2) waiting for ACK, NACK and retransmission petitions to be inserted in the ack_queue and data_queue respectively by the receiver. The sender logic is implemented in the following pseudo-code:

**sender:**

```plaintext
initialization:
   exp := 1
   seqnum := 1
   shared bk_list := {}
   shared data_queue := {}
   shared ack_queue := {}

procedure send_data(m, obj)
   obj.sn := seqnum
   seqnum := seqnum + 1
   if obj is in bk_list:
      remove(obj, bk_list)
      add_tail(obj, bk_list)
   else
      add_tail(obj, bk_list)
      send(m, obj.sn)

procedure send_ack(m)
   seqnum := seqnum + 1
   send(m, seqnum)

do forever:
   for each {m, obj} in data_queue:
      broadcast(m, obj)
      remove({m, obj}, data_queue)
   for each m in ack_queue:
      send_ack(m)
      remove(m, ack_queue)
```

The receiver is based on two ideas: messages received are always delivered even if they are out of sequence, and only the latest state change is inserted in the bk_list, so we do not keep the complete sequence of state changes but only the last state reached. These ideas are extracted from the CTS semantics since 1) we assume that there is no dependency between states that talk about different flows and 2) we can easily identify more recent states changes for they always have higher sequence number than the older ones. If the receiver sees an ACK, all the confirmed messages are removed from the bk_list. If a NACK message is received, omitted state changes are extracted from the bk_list and are inserted in the data_queue to be transmitted by the sender. The receiver part can be implemented as follows:

**receiver:**

```plaintext
initialization:
   exp := 1
   seqnum := 1
```
queue := {}
shared bk_list := {}
shared data_queue := {}
shared ack_queue := {}

procedure enqueue_data(m)
add((m, obj), data_queue)

procedure enqueue_ctl(type, exp, rcv):
m.type = type
m.from = exp
to := seqrcv - WINDOW_SIZE
add({m, obj}, data_queue)

when receive(m, seqrcv):
if seqnum <> exp:
    window := WINDOW_SIZE
    enqueue_ctl(NACK, seqrcv, exp)
if --window = 0:
    window := WINDOW_SIZE
    from := seqrcv - WINDOW_SIZE
to := seqrcv
    enqueue_ctl(ACK, from, to)
if is_ack(m):
    for each obj in bk_list:
        if obj.sn >= m.from and
           obj.sn <= m.to:
            remove(obj, queue)
if is_nack(m):
    for each obj in bk_list
        if obj.sn >= m.from and
           obj.sn <= m.to:
            enqueue_data(m, obj)

deliver(m)

As said earlier, the replication protocol also introduces the use of selective positive acknowledgement (ACK) mechanisms to confirm the correct delivery and reduce the size of the backlog list. Since we use a list (search operations to retransmit messages are 0(n)) the selective acknowledgement mechanism guarantees that the bk_list size is reduced, thus decreasing search time. Of course, a more efficient data structure can be used to get better search timing but we consider this discussion out of the scope of this work. We assume that the ACK window size is a tuneable parameter. Intuitively, if the window size is small, the number of acknowledgments sent increases but the retransmission queue is reduced. Due to space restrictions in this work, we leave the best window size value for future improvements.

If the receiver notices message omission situations, it inserts a NACK in the ack_queue that indicates the range of lost messages, i.e., enqueue an explicit request to resend messages from a certain sequence \( a \) to another sequence \( b \). Observe that the number of messages \( n \) resent by the sender holds \( n \leq b - a \) since the messages that contains superseded state changes for a certain flow are not retransmitted. Although the message omission scenario could seem an unrealistic situation, it may happen due to overruns in the sender/receiver buffer since our protocol has no congestion control mechanisms. Particularly, message omission cases are linked to heavy stress scenarios in which the sender transmits more messages than the receiver can handle. Nevertheless, our protocol reduces the number of state changes that need to be retransmitted under these situations.

To conclude, we present an example case of the behaviour of our protocol under message omission (Fig. 3) where messages with sequence numbers \( seq + 1 \) and \( seq + 2 \) are omitted. Such messages contain state changes \( S_1 \) and \( S_2 \) regarding flow \( F_1 \) respectively. However, sequence number \( seq + 3 \) regarding flow \( F_2 \) is correctly delivered to the backup replica firewalls. This backup replica notices that two messages got lost, and thus, explicitly requests messages from \( seq + 1 \) to \( seq + 2 \) by means of a NACK. Then, the sender side notices that both messages contained two state changes regarding the same flow \( F_1 \). So, it sends only one message with sequence number \( seq + 3 \) containing the last state \( S_2 \) reached by the connection \( F_1 \). Therefore, no old messages that contain superseded states are resent. As said, this behaviour differs from other reliable protocols such as TCP that would retransmit every lost message.

![Fig. 3. Message omission situation](image)

IV. Evaluation

To evaluate FT-FW, we have implemented a state proxy daemon for stateful firewalls, the so-called conntrackd (connection tracking daemon) [10]. This software is a userspace program written in C that runs on Linux. We did not use any optimization in the compilation.

The testbed environment is composed of AMD Opteron dual core 2.2GHz hosts connected to an 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 (Fig. 4). We have adopted a Primary-Backup configuration for simplicity. Thus, FW1 acts as Primary and FW2 acts as Backup. In order to evaluate the solution, we reproduce a very 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 client HTML suite have been used.

**A. CPU overhead**

We have measured CPU consumption in FW1 and FW2 with and without full state replication. The tool cyclesoak [11] has been used to obtain accurate CPU consumption measurements. For the sake of simplicity, we use a reduced version of the graph of TCP states (Fig. 5).

Since each TCP connection has 6 possible state changes, each HTTP request generates up to \(6 \times \text{total number of requests}\). The results obtained in the experimentation have been expressed in a graph (Fig. 6). In the primary, the full state replication is CPU consuming, reaching up to 42.7% of CPU load. Although this only means 25% more than without replication. On the other hand, the backup consumes 11% of CPU due to the state replication - the backup does not filter packets. 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) these days. Thus, we can conclude that FT-FW guarantees the connection recovery at the cost of requiring CPU power.

**B. Round Trip**

In order to obtain the delay that FT-FW introduces in client responses, we have created up to 1700 HTTP GET requests per second while measuring 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 the following table (in milliseconds):

<table>
<thead>
<tr>
<th>HTTP GET rps</th>
<th>w/o replication</th>
<th>w/ replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>238.5</td>
<td>246</td>
</tr>
<tr>
<td>1700</td>
<td>243</td>
<td>247</td>
</tr>
</tbody>
</table>

As we can observe, the increment in the round trip time is between 4 and 7 milliseconds so that we can say that the delay introduced in clients responses is negligible.

**C. Recovery Time**

We have also measured the time needed to inject SOs that represents connections from the proxy daemon’s external cache to the in-kernel CTS. The results (Fig. 7) show that the injection of 20000 state objects is close to 180 ms, which is an affordable delay. Another obvious conclusion extracted from the results obtained is that the time required to inject objects increases linearly.

We can conclude from the results that FT-FW requires extra CPU power in order to improve CSD of CBSF but has a negligible impact in terms of client response time.

**V. RELATED WORK**

Several works have proposed fault tolerance solutions for web servers. Such approaches can be applied to CBSF. In
several drawbacks: proposed in such work is to migrate the states to the backup to reduce the overhead introduced by the protocol stack. The solution theoretically permits low latency networking and decreases latency of another without involving the protocol stack, which propagation by moving data from the memory of one replica [17]. This approach is not protocol dependent and allows stateory Access (RDMA) mechanisms have already been presented [13]. This solution is client-transparent, it just requires server side modifications, and does not require extra hardware. The main idea consists of a backup TCP (BTCP) stack implementation that is a silent version of TCP. The web servers, back-ends, are organized into a ring. Each time a request is received, the balancer sends forwards the request to the primary server as well as to the backup. The backup keeps a backup socket structure that contains the request state. Since the BTCP stack never sees incoming traffic, some important information has to be inferred, so the authors propose a complete logic to infer such information. Also two improvements are proposed. The evaluation is done in a 100 Mbit network. This solution has similarities with fault tolerant TCP approaches such as FT-TCP [16] whose key concept are the loggers, a software component that stores in the backup replica all the packets processed by the primary replica via network tapping techniques. In case of failure, the backup replays the complete communication until the last consistent state before the failure is reached. This solution may be applied to the scenario described in this work. However, it does not scale well for long connections with high data exchange rates. We must take into account that long established communications that involves the exchange of tons of packets that have to be stored by the logger may require long time to recover since the logger would need to replay all the packets to reach the last connection state. For our purposes, this is not required since we only need the last state reached to recover a connection. FT-TCP approaches are stuck on the Primary-Backup setting, thus load balancing cannot be easily deployed, and are too protocol centric (they only enforce TCP recovery). Moreover, the authors of FT-TCP do not cover the loggers’ machine failure.

A state replication solution based on Remote Direct Memory Access (RDMA) mechanisms has already been presented [17]. This approach is not protocol dependent and allows state propagation by moving data from the memory of one replica into that of another without involving the protocol stack, which theoretically permits low latency networking and decreases the overhead introduced by the protocol stack. The solution proposed in such work is to migrate the states to the backup replica on the event of a fault. However, this approach has several drawbacks:

- The solution assumes that the failure does not happen in the memory space that stores state information.

- The architecture proposed requires a new piece of hardware (an intelligent card) that incurs an extra cost that needs to be assumed.

- The porting of the existing high availability solutions to work on such hardware would also be required.

- RDMA mechanisms have been specifically designed for high performance computing (HPC). However, the technology may result intrusive in the scenario described in this work.

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. A Primary/Backup setting is presented. Such solution is based on an event-driven model and describes the Stateful Networking Equipments (SNE) library. Such library extends existing HA solutions to enable highly available firewalls and routers. Similarly, the authors of [19] have proposed an event-driven fault-tolerant solution for VoIP which implements an HA middleware to save or restore states on the event of failures.

VI. DISCUSSION

The FT-FW solution does not guarantee that firewall replicas contain the same set of states at any time, i.e. we cannot guarantee that replicas are one-copy equivalences as in database fault-tolerant solutions. Although database transaction schemes provide CSD of 1 for all states, they are too heavyweight for the scenario covered in this work since it would roughly degrade clients responses. Our target is to improve CSD without harming performance.

Nevertheless, since replicas are not one-copy equivalences, we do not know how many state changes were not successfully propagated. Consider the following example:

1) A packet triggers a state change. Thus, the state variable associated to the flow $F_i$ is updated: $S_n \rightarrow S_{n+1}$.

2) The state change of $F_i$ from $S_n \rightarrow S_{n+1}$ sends through an event to the SP. However, a failure happens before the state proxy propagates the state change through the dedicated link to other replicas.

3) Backup firewalls remain in state $S_n$ but the flow $F_i$ is in state $S_{n+1}$ in the primary replica. Thus, the backup would not be able to recover the flow $F_i$.

The problem detailed above is related to the asynchronous nature of FT-FW. As said, although synchronous solutions (such as database transaction schemas) guarantee the one-copy equivalence property, the performance drop incurred would be unaffordable. So, to estimate the number of state changes that may not be replicated due to a failure, we can calculate the rate of out of date state objects in the backup replicas with the following formula:

$$\theta = \frac{\text{NIC tx ring size (in packets)}}{\text{number of states objects}}$$

Let’s assume a hardware failure in a busy primary firewall which is filtering 20000 flows, and whose NIC transmission ring is of 1000 packets. If the tx ring is full during the failure, i.e. the ring holds 1000 packets with state changes
The solution proposed is not dependent on the failure detection and recovery algorithms guaranteed, and no extra hardware is required. It guarantees simplicity. Also, fast client responses and quick performance. The architecture follows an event-driven model (Tolerant Firewall) that has negligible impact in terms of resource consumption.

For cluster-based stateful firewalls known as FT-FW (Fault Tolerant FireWall) that have negligible impact in terms of performance. The architecture follows an event-driven model that guarantees simplicity. Also, fast client responses and quick recovery are guaranteed, and no extra hardware is required. The solution proposed is not dependent on the failure detection schema nor the layer 3 and 4 protocols that the firewalls filter.

The FT-FW replication protocol improves CSD since the sender never stops sending messages and the receiver handles all messages (even those that are out of sequence). The protocol reduces the number of retransmitted messages under message omission situations so that it scales better than any other existing transport protocol for CBSF.

As future works, we are dealing with several optimizations to reduce CPU consumption without harming CSD in environments with limited resources such as embedded systems. We also plan to study multi-primary scenarios in which more than one firewall filters traffic at the same time.

VII. Conclusion and Future Works

In this work we have proposed a fault-tolerant solution for cluster-based stateful firewalls known as FT-FW (Fault Tolerant FireWall) that has negligible impact in terms of performance. The architecture follows an event-driven model that guarantees simplicity. Also, fast client responses and quick recovery are guaranteed, and no extra hardware is required. The solution proposed is not dependent on the failure detection schema nor the layer 3 and 4 protocols that the firewalls filter.

The FT-FW replication protocol improves CSD since the sender never stops sending messages and the receiver handles all messages (even those that are out of sequence). The protocol reduces the number of retransmitted messages under message omission situations so that it scales better than any other existing transport protocol for CBSF.

As future works, we are dealing with several optimizations to reduce CPU consumption without harming CSD in environments with limited resources such as embedded systems. We also plan to study multi-primary scenarios in which more than one firewall filters traffic at the same time.

Acknowledgment

We would like to thank to Alberto Escudero-Pascual from IT+46 for his worthy suggestions. 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).

References