CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Formal Verification of FlowSync

Applying Formal Methods to a Distributed System for Managing Asymmetrically Routed Internet Traffic

Master's thesis in Computer science and engineering

Frans Bergman
Shubhankar Choudhari

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Formal Verification of FlowSync

Applying Formal Methods to a Distributed System for Managing Asymmetrically Routed Internet Traffic

Frans Bergman
Shubhankar Choudhari

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Formal Verification of FlowSync
Applying Formal Methods to a Distributed System for Managing Asymmetrically Routed Internet Traffic
Frans Bergman
Shubhankar Choudhari

Supervisor: Dr.Yehia Abd Alrahman
Advisor: Mr.Daniel Romell
Examiner: Prof.Wolfgang Ahrend

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Formal Verification of FlowSync
Applying Formal Methods to a Distributed System for Managing Asymmetrically Routed Internet Traffic
Frans Bergman
Shubhankar Choudhari
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Complex network layouts, currently used by internet service providers and large corporate networks, have lead to a phenomenon known as asymmetric routing. That is, the two directions of a traffic flow take different paths through the network. This poses a challenge for network management systems, which typically require knowledge of both directions of traffic to perform classification and statistics collection. In this thesis, we apply formal methods to identify and suggest solutions to issues in such a system. The system is called FlowSync, and has been developed at Sandvine for use in its internet traffic management product. The latter is an international company providing systems for classifying and managing internet traffic. We model three distinct components of FlowSync; classifier sharing, statistics synchronization and link redundancy, and identify potential issues and limitations in all three. Two of these limitations are consequences of known problems in distributed system design, and the relation to these problems is discussed, as well as known techniques for working around them.

Keywords: Formal methods, model checking, distributed systems, internet traffic, asymmetric routing.

Acknowledgements

We want to express our profound appreciation to all who provided us with the possibility to complete this thesis. We wish to express our sincere gratitude to our university supervisor, Dr. Yehia Abd Alrahman, for his valuable guidance, consistent support, and patient encouragement throughout our project. His broad knowledge and expertise in the subject matter significantly contributed to our work. We would also like to thank our examiner, Prof. Wolfgang Ahrendt. His critical and constructive evaluation of our work has helped us improve the quality of our thesis.

We thank our company supervisor, Mr. Daniel Romell, for his instrumental advice, continuous assistance, and insightful critiques during the project. His professional approach and commitment to excellence substantially enriched our project experience.

In addition, we want to thank our manager, Mr. Kenneth Johansson, for his encouragement and all the opportunities we were given to further our knowledge and research. His unwavering confidence in us inspired us to meet all challenges head-on.

Finally, we wish to express our gratitude to our colleagues and friends for their understanding and support in completing this project. Completing this work is not only a product of our effort but also the encouragement, support, and guidance of numerous individuals who made this journey an enriching learning experience.

Frans Bergman, Gothenburg, 2023-06-20
Shubhankar Choudhari, Gothenburg, 2023-06-20

Contents

List of Figures

List of Tables

Glossary

5-tuple A unique combination of source and destination address pair, source and destination port pair, and transport protocol. Used to identify TCP/IP flows. 6, 7

checksum A checksum is a small block of data used to validate the integrity of a larger block of data. The checksum is produced using a deterministic checksum function. A checksum is typically used to verify the integrity of packets transferred over the internet [1], [2]. 3

datagram A datagram is the smallest data transfer unit in connectionless communication, typically using the User Datagram Protocol. Each datagram has two sections: header and data. The header contains the source and destination port, the length of the message, and a checksum [1]. 3

flow A bidirectional stream of IP packets. This is a general term for TCP connections and UDP conversations. 6, 7, 9, 11, 16

heartbeat Heartbeat packets are messages that are sent periodically between network nodes to notify each other of their presence on the network [3]. 10, 11, 31, 39, 41, 43, 58

port Port numbers are used in the TCP and UDP protocols for service identification and connection multiplexing. Every segment or datagram has a source and destination port field [1], [2]. 3, 6

segment The smallest unit of data transmitted in a TCP stream [2]. 3, 13

Acronyms

CPU Central Processing Unit. 58

DNS Domain Name System. 3, 5

DoS Denial of Service. 31

FIFO First In, First Out. 34

FTP File Transfer Protocol. 3

HTTP Hyper Text Transfer Protocol. 3, 7, 8, 46, 48, 49, 54

IP Internet Protocol. xi, 2, 3, 6, 9

LAN Local Area Network. 9

LTL Linear Temporal Logic. xi, xii, 25–27, 30, 33, 34, 36, 38, 41, 49, 50

MAC Medium Access Control. 3, 13

NIC Network Interface Card. 36

RTT Round-Trip Time. 12, 33, 45, 47

SDN Software Defined Networking. 27, 28, 59

SMTP Simple Mail Transfer Protocol. 3

SSL Secure Sockets Layer. 26

TCP Transmission Control Protocol. xi, 2–7, 12, 13, 24, 28, 32–34, 36, 46, 49, 51, 55

UDP User Datagram Protocol. 3, 5, 24

VoIP Voice over Internet Protocol. 1, 24

1

Introduction

As internet traffic bandwidth grows every year [4], so does the task of managing said traffic. This calls for systems that can efficiently classify internet traffic in order to shape, filter, and collect statistics. These systems can improve the network's overall efficiency by identifying bottlenecks, improve the Quality of Experience of latency-critical services such as Voice over Internet Protocol (VoIP), and enforce policies, for example, blocking illegal or potentially dangerous content such as malware.

Sandvine is an international company that provides combined hardware and software systems to its customers for managing network traffic. Its target customers range from enterprises to governments and internet service providers. As the bandwidth handled by these large entities grows to the order of tens of terabits per second [5], [6], even the most powerful hardware platforms fall short of handling all of the throughput alone.

To mitigate the bottleneck of using a single hardware node, horizontal scaling can be employed by distributing the load of classifying, shaping traffic, collecting statistics, etc., across several nodes. These nodes will be referred to as Traffic Processors, abbreviated as "TPs". When load balancing traffic across TPs using off-the-shelf routers, a phenomenon referred to as "asymmetric routing" emerges[1]. That is, the two directions of a traffic flow between two peers take different paths through the network [7]. What this means in the context of traffic management is that each TP only has partial information about each flow. To solve the problem of partial information and allow traffic management in an asymmetrically routed network environment, Sandvine has developed a metadata-sharing protocol called FlowSync. This thesis project will cover the application of formal methods to components of the FlowSync protocol in order to identify faults and limitations in their design and implementation.

[1] For the purposes of this thesis, internet traffic that is subject to asymmetric routing will be referred to as *asymmetric traffic*.

The structure of this report is as follows: Section 1.1 gives an overview of internet traffic and the internet protocol suite and Section 1.2 introduces traffic classification. Section 1.3 describes the FlowSync protocol and the problems it aims to solve. Chapter 2 presents the problem statement and outlines the scope of the thesis. Chapter 3 provides a background of relevant research done in this field. Chapter 4 outlines the method used to produce the results. Chapter 5.1 presents a survey of modeling frameworks. Chapter 6.1 presents the model of FlowSync's link redundancy component and discusses its results. Chapter 7.1 presents the model of FlowSync classifier sharing and statistics synchronization components and discusses its results. Finally, Chapter 8 concludes the thesis and gives suggestions for future work.

1.1 Internet traffic and protocols

Internet traffic makes use of the internet protocol suite, also known as TCP/IP, to facilitate worldwide networked communication [8]. The suite is divided into 4 layers: The Link layer, the Internet layer, the Transport layer, and the Application layer. Each layer builds upon the *services* provided by the lower layer(s) and provides services to the layers above. This conceptual stacking of layers provides separation of concerns since each layer can be reasoned about separately, without directly relying on the implementation details of the other layers [9].

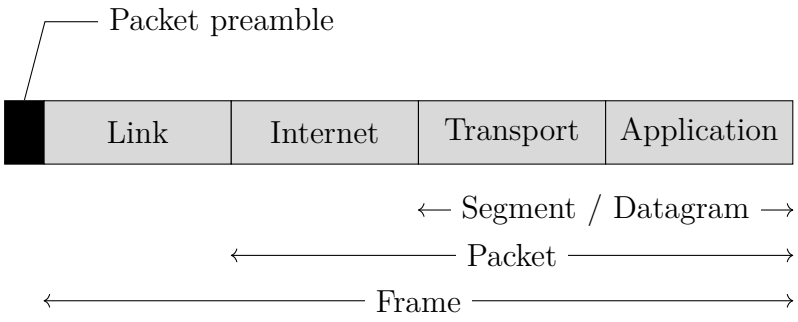

Figure 1.1: Structure of a complete TCP/IP message.

The *link layer* is the layer closest to the hardware used to transport traffic. This layer provides transportation of *frames* from one device to another over some medium. For example, Ethernet for wired connections or IEEE 802.11 for wireless ones.

The *internet layer* facilitates routing of *packets* from one device on the internet to another. Since not all devices on the internet have a direct link to each other, this may require *routing* the packets through multiple devices, referred to as *routers*. Routers inspect the *IP address* of packets to determine where to send them next until they reach their destination.

The *transport layer* builds upon the internet layer to provide additional services, such as integrity verification and reliable transport. Messages in the transport layer are referred to as segments or datagrams, depending on whether Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) is used. The majority of internet traffic, in terms of bandwidth, uses one of these two transport layer protocols [10]. Both protocols provide integrity verification using checksums, as well as application multiplexing using port numbers. The main difference is that TCP is a stateful protocol providing reliable transport and congestion control over an established TCP connection, whereas UDP is stateless, provides no additional reliability over the underlying layers, and provides no congestion control mechanism.

The *application layer* is the highest layer in the TCP/IP model. This is where application protocols such as HTTP, FTP, DNS, SMTP are implemented. In this report, HTTP will be used to illustrate examples of internet traffic, as it (a) is common, (b) uses clear text for data transfer, and (c) uses the TCP transport layer protocol.

Figure 1.1 shows the structure of a TCP/IP message, including the header associated with each layer. The link level header typically contains source and destination MAC addresses, while the Internet header contains IP addresses, and the transport header contains port numbers, a checksum and sequence numbers. Application data is entirely determined by the application-level protocol used.

Transmission Control Protocol (TCP) is a connection-oriented protocol. A connection between the client and the server is established before data can be sent. To establish this connection, a three-way handshake is used.

Figure 1.2 shows a complete TCP conversation, including the three-way (or 3-step) initialization handshake and termination handshake.

When a client wants to connect to a server, it sends a synchronize (SYN) message to it. This message also contains the client's sequence number to let the server know what sequence numbers to expect in the following messages.

When the server receives the SYN message, it responds with a synchronize-and-acknowledge (SYN + ACK) message. The SYN sequence number is that of the server, while ACK is the client's sequence number + 1.

The client sends ACK to the server on receiving SYN + ACK from the server. Again, ACK is the server's sequence number + 1. Now, the sequence numbers to be used in each direction have been agreed on. Once the three-way handshake is complete, the TCP connection can be used for data transfer.

The sequence numbers ensure that datagrams are in order. If a datagrams goes missing, the TCP implementation re-sends it without the involvement of the application layer.

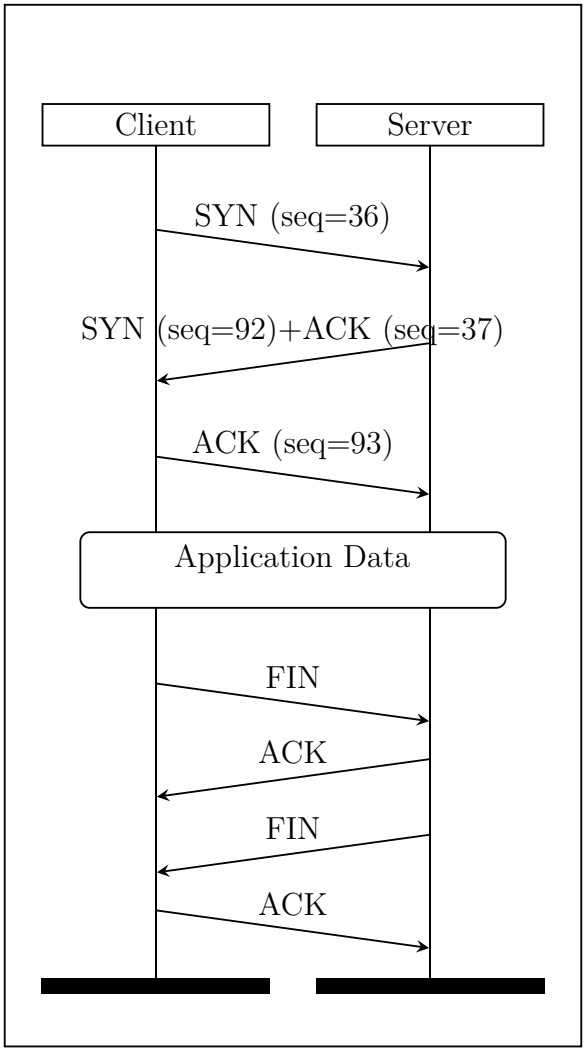

Figure 1.2: TCP connection initialization and termination.

When the application data transfer is complete, the connection is closed by exchanging finish (FIN) and ACK messages in a four-way handshake. The reason for not using a three-way handshake, similar to the initialization handshake, is that each end of the connection is closed separately, allowing each peer to finish sending its data before closing the connection [2].

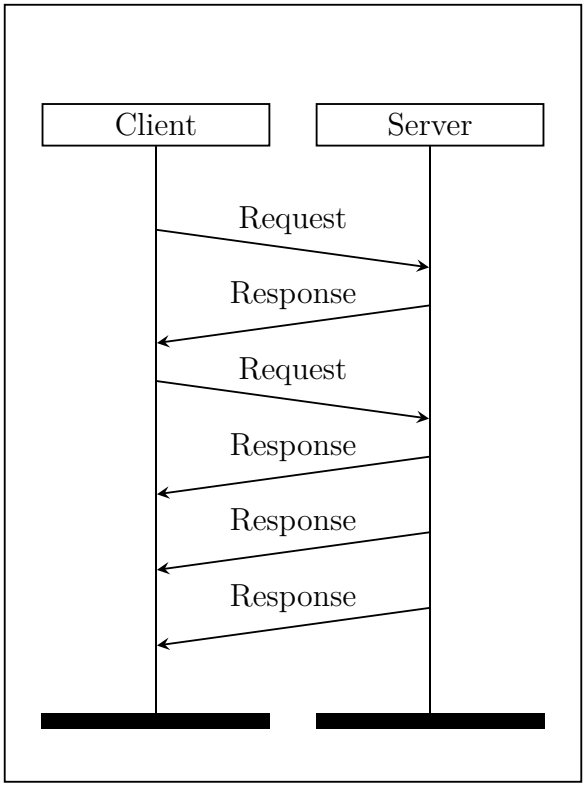

Figure 1.3: UDP data transfer.

User Datagram Protocol (UDP) is used for latency-sensitive transmissions such as video playback and DNS lookups. Latency is reduced by not explicitly establishing a connection before data transfer, as in TCP.

Unlike the TCP's three-way handshake, in UDP, a client can send data directly to the server as shown in Figure 1.3. This creates advantages and disadvantages for the transmission. The advantage is that this speeds up communication; however, due to the absence of sequence numbers, there is no way to know whether the received packets are received in order, duplicated, or lost.

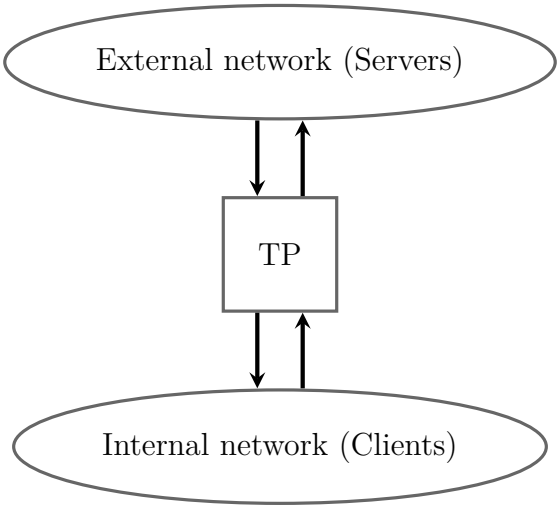

Figure 1.4: An example Traffic Processor deployment. The TP acts as a middle-box, forwarding packets between clients on the internal network and servers on the external network.

1.2 Classifying internet traffic

As previously stated, one of the tasks of a Traffic Processor is to classify traffic. This section introduces the classification of internet traffic in the absence of asymmetric routing. Section 1.3.2 will then explain how FlowSync is used to handle classification of asymmetric traffic.

Internet traffic can be reasoned about not only in terms of packets but in terms of packet *flows*. Flow is a general term that refers to a conversation between two peers on a network. The two peers involved in a flow are typically referred to as the *client* and the *server*, where the client is the one that initiates the flow by sending the first message. The stream of packets that are sent from the client to the server is referred to as the upstream. The opposite direction is referred to as the downstream.

Each packet of internet traffic belongs to a flow, identified by its TCP/IP 5-tuple. That is, its source and destination IP address, its source and destination port, and the transport protocol used. Analyzing only surface-level metadata, such as transport protocol and port numbers, may not be enough to accurately differentiate services [11]. Because of this, accurate classification of a traffic flow is only possible by inspecting the contents of its packets. The TP is placed on the network path between the clients and servers to facilitate this. A typical deployment is shown in Figure 1.4, where the TP acts as a middle-box, forwarding packets between two networks.

A TP may be processing many flows concurrently. In order to keep track of the state of each flow, a table is used to hold metadata for each flow. This table is indexed by the flow's 5-tuple, which allows the TP to associate every traffic packet it observes with its corresponding metadata entry.

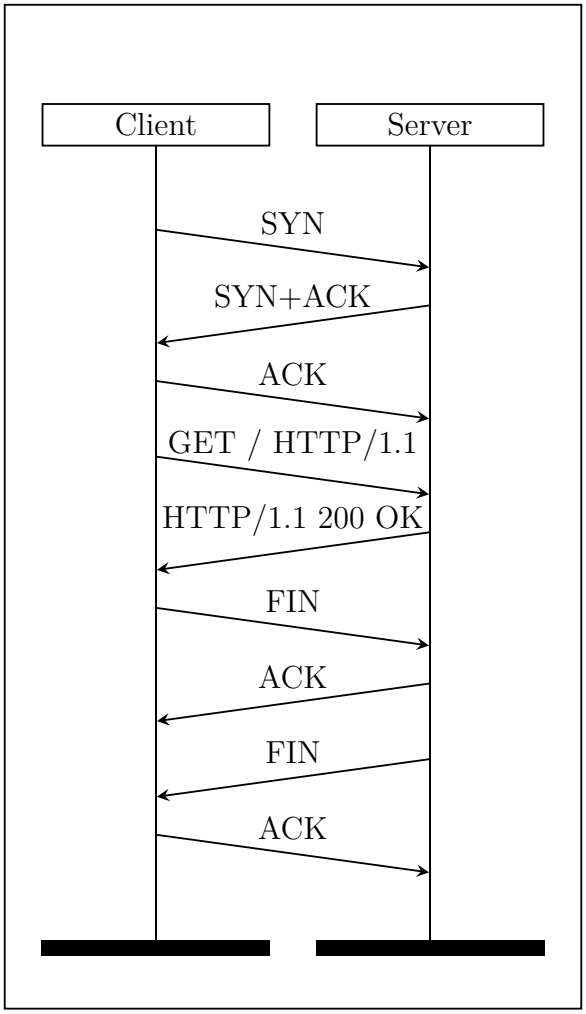

Figure 1.5: Example HTTP conversation.

A central component of the classification engine is the *classifier state machine*. One such state machine exists for each flow, and the state of each machine is stored together with the flow's metadata in the connection table. This state machine is used to determine the type of each flow by feeding it the contents of the packets observed. When a packet belonging to a certain flow is seen, the next state of the classifier state machine is computed, and the connection table is updated.

An example of a traffic flow is a TCP connection that fetches a web page over HTTP. Consider the HTTP conversation shown in Figure 1.5 taking place between a server on the internet and a user on the internal network. Classification of this connection takes place as follows:

1. The TP observes the TCP SYN packet from the client to the server and initializes a new classifier state machine associated with this connection's 5-tuple.
2. The SYN+ACK and ACK packets are observed, and the connection is now classified as an active TCP connection.

3. The HTTP GET request is observed. The connection is now classified as HTTP/1.1.
4. The HTTP 200 response is observed. The connection is now classified based on the contents of the response. For example, its content type and encoding are now known.
5. The 4-step termination handshake is observed, and the flow metadata, including the classifier, are removed from the connection table.

1.3 FlowSync

As previously mentioned, asymmetric routing may occur when load balancing IP traffic across multiple Traffic Processors. To allow the classification of asymmetric traffic, Sandvine has developed a protocol for sharing traffic metadata across TPs. The protocol specifies how TPs share metadata about traffic flows over a Local Area Network (LAN), referred to as a *domain*. This includes messages for broadcasting that a new flow has been established, updates about the state of the classifier for a flow, flow statistics, and notifying peers that a flow has ended. This protocol is referred to as *FlowSync*.

FlowSync is used to coordinate the management of millions of connections across over 100 Traffic Processors in a single FlowSync network. As of writing, Sandvine's highest-capacity hardware platform, the iQ52600, supports up to 150,000,000 concurrent connections [12].

Figure 1.6 shows the simplest possible deployment using FlowSync, with two TPs handling fully asymmetric traffic, i.e., all traffic in each direction is routed to the same TP. The FlowSync domain is illustrated by a bi-directional arrow between the TPs. A pair of upstream and downstream links connected to a TP is referred to as a *traffic channel*.

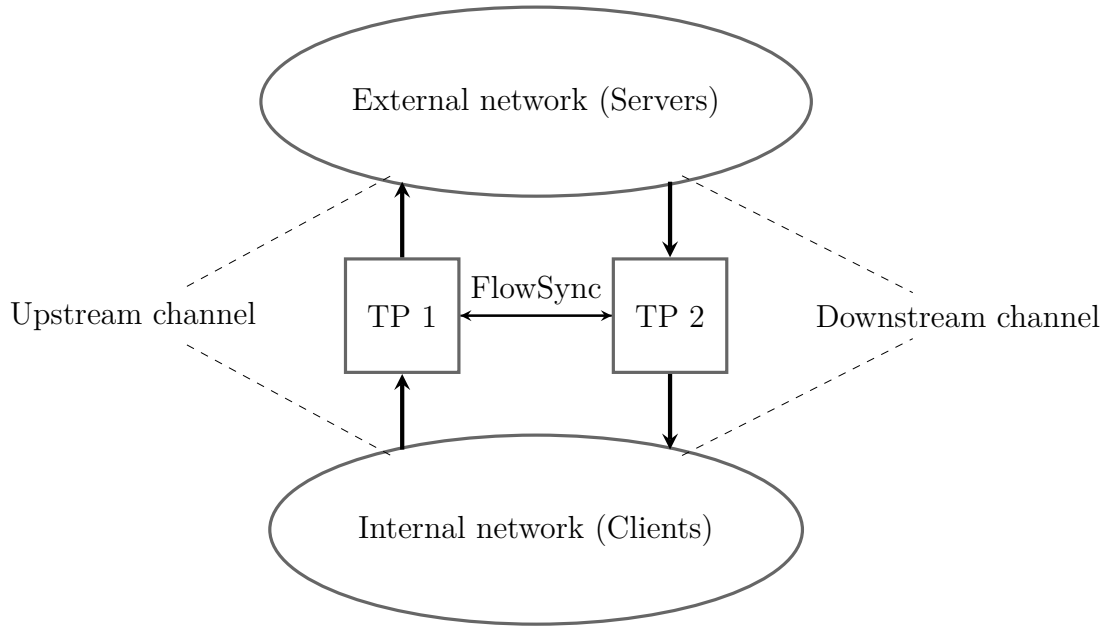

Figure 1.6: The simplest possible FlowSync deployment, using two TPs to handle fully asymmetric traffic.

Conceptually, FlowSync consists of three major components; classifier sharing, statistics synchronization, and link redundancy. Figure 1.7 illustrates the relationship between the components, with the link redundancy layer providing message transport to the classifier sharing and statistics synchronization components in the service layer above. The following subsections outline the intended behavior of each of the three components of FlowSync.

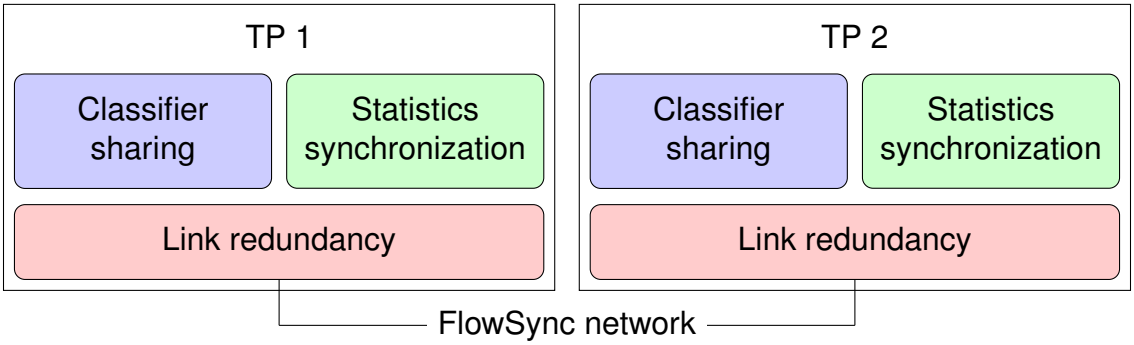

Figure 1.7: High-level overview of FlowSync components

1.3.1 Peer discovery and link redundancy

The set of peers present on a FlowSync network does not need to be defined manually. Rather, peers are discovered dynamically using broadcast heartbeat packets and unicast response packets. Heartbeat packets are sent at regular intervals by each TP to announce its presence on a domain.

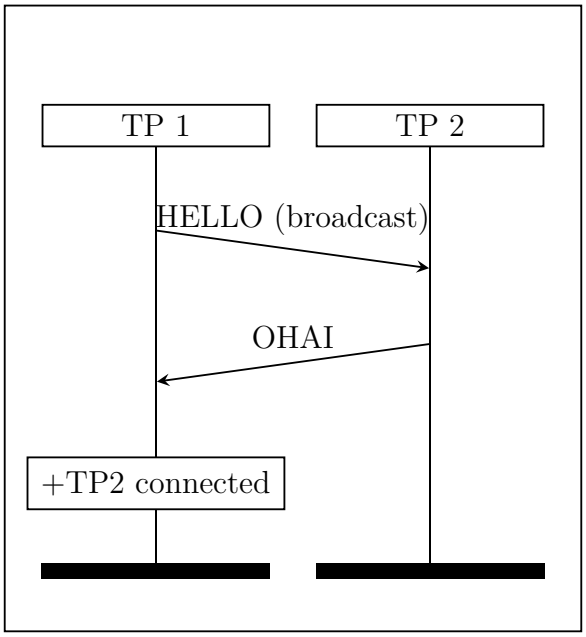

Figure 1.8: FlowSync peer discovery

Figure 1.8 shows how the messages HELLO and OHAI are used to discover peers on a FlowSync domain. The conversation takes place as follows:

1. HELLO is broadcast by TP 1 to announce its presence on one of its domains.
2. OHAI is sent in response to the HELLO packet. This packet is sent as unicast to TP 1.
3. TP 1 receives the OHAI packet, which means that TP 1 is now aware that there is a bi-directional link to TP 2 on this domain.

This way, each TP maintains a list of all TPs available based on the OHAI packets it receives. A timer is used in each TP to remove any peer who has not been seen for a set amount of time.

In order to facilitate redundancy and load balancing on the FlowSync network, FlowSync traffic[2] may be distributed across several domains. A set of domains connecting a set of TPs is referred to as a FlowSync *network*. Let us consider a scenario with 3 TPs (TP1, TP2, and TP3) that communicate on 2 domains (A and B). In practice, each domain would typically be realized using one or more network switches.

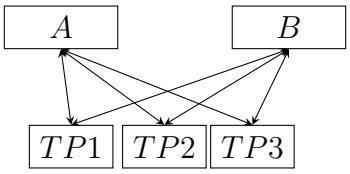

Figure 1.9: Working FlowSync domains.

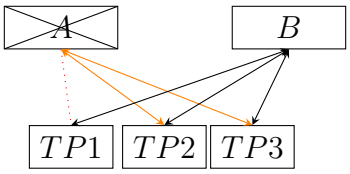

Figure 1.10: FlowSync domains with a broken link.

Scenario 1: All links are working, shown in Figure 1.9.
Load balancing is achieved by distributing the FlowSync traffic across the domains on which all TPs have been discovered by the heartbeat procedure.

Scenario 2: The link between TP1 and domain A fails, shown in Figure 1.10.
Since TP1 has lost its link to domain A, all other TPs also stop using domain A in order to ensure that the FlowSync packets addressed to TP1 are not lost.

1.3.2 Classifier sharing

In an asymmetric environment, the upstream and downstream flow directions are seen by two different TPs. This means that the TPs involved need to share classification metadata with each other. For the sake of simplicity, the state of a flow's classifier can be seen as a large integer, which can change any time a packet is seen in either direction. With this simplification, for a flow to stay synced between two TPs, each TP needs to send a copy of the classifier state integer to its peer every time it changes.

The following packets are defined for classifier sharing over FlowSync.

[2]Note that this is the traffic used to share information between TPs, not to be confused with internet traffic.

- SEEN: A TP broadcasts this packet when it observes a Transmission Control Protocol (TCP) SYN/ACK packet on its traffic channel without first seeing the TCP SYN packet. This marks the start of a new asymmetric connection as the TCP SYN packet must have been seen by a different TP.
- UPDATE: This packet lets the receiving TP know about changes needed for identifying the connection, i.e., the updated classifier state.
- ENDED: This packet informs the receiving TP that the connection has ended, typically because a TCP FIN packet was seen on a traffic channel.
- FORGET: Sent when an unrecoverable error occurs, indicating that a connection is out-of-sync.

A prerequisite for the classifier state machine to operate correctly is that it is updated in the order that packets are exchanged between the client and the server. This is not a problem when there is only a single Traffic Processor with a single copy of the classifier. However, when two TPs share the classifier state, this means that an up-to-date copy of the classifier must be available as soon as the next packet (in either direction) is seen. In practical terms, this means that the Round-Trip Time of the FlowSync network must be shorter than that of the server-client pair.

In order to identify when packets on the FlowSync network are dropped or reordered, sequence numbers are employed. These sequence numbers increment whenever the classifier state changes or a sequence-numbered FlowSync UPDATE is received. If an UPDATE with an unexpected sequence number is received, the connection is marked as out-of-sync, and the FORGET message is sent to the FlowSync peer.

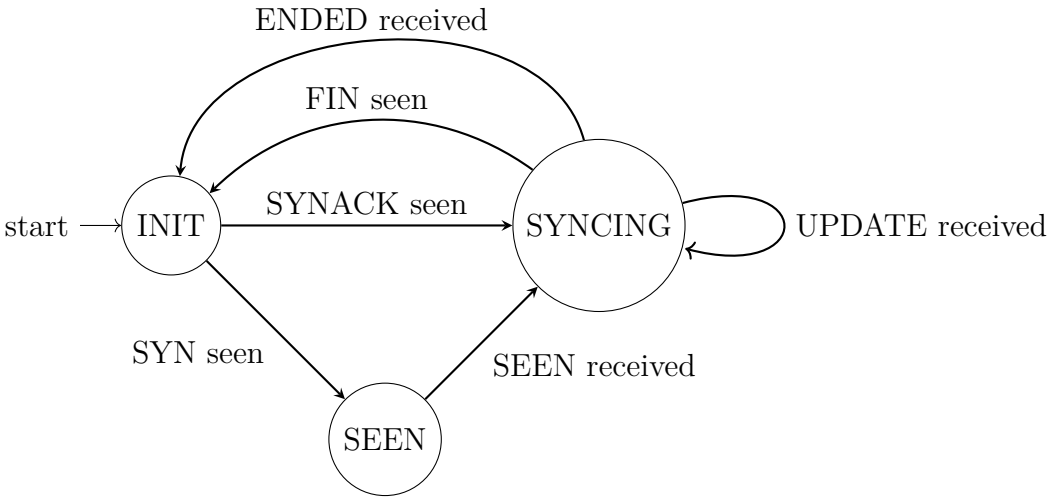

Figure 1.11: State diagram for a connection on a TP performing classifier sharing.

Figure 1.11 shows a state diagram of the classifier sharing algorithm, while Figure 1.12 shows the messages exchanged in a scenario with 3 TPs. The scenario is summarized in the following steps:

1. TP1 observes the SYN packet on its traffic channel and moves into state SEEN.

2. After some time, TP2 observes the SYN/ACK packet on its traffic channel. As it has not previously seen the SYN of this connection and is still in state INIT, TP2 broadcasts a SEEN packet on the FlowSync network and moves to state SYNCING.
3. TP1 and TP3 receive the broadcast. Since TP1 saw the SYN earlier and is in state SEEN, it sends an UPDATE packet (seq=0) to TP2 to acknowledge that it has seen the other half of the connection and moves to state SYNCING. TP3 ignores the broadcast, as it has not previously seen this connection.
4. Once TP2 receives the UPDATE packet from TP1, FlowSync is active, and both sides have each other's MAC address.
5. TP2 sees a TCP segment belonging to the connection on its traffic channel and sends an UPDATE packet (seq=1) to TP1, containing a copy of the updated classifier state integer.
6. TP1 receives the UPDATE packet and updates its classifier state copy accordingly, remaining in state SYNCING.
7. Steps 5 and 6 are repeated with either TP1 or TP2 observing TCP segments and communicating the state changes to the other TP.
8. In the end, when TP1 sees a TCP FIN packet, it sends ENDED to TP2 to mark the end of the connection and moves into state INIT.
9. When TP2 receives the ENDED message and moves back to the INIT state.

In reality, not all packets on the FlowSync network reach their destination in the intended time, causing the classifier sharing to go out of sync. Figure 1.13 shows an example of this scenario. The UPDATE(seq=2) packet from TP2 is delayed and collides with the next UPDATE(seq=2) packet coming from TP1. On realizing that the connection is out of sync, TP2 sends a FORGET packet to TP1, which in turn sends an ENDED packet to mark the end of the communication.

In a different scenario, it may happen that after broadcasting SYN/ACK, TP2 receives multiple SEEN packets. This means that two different TPs have observed the SYN for that connection, indicating a misconfiguration of the traffic network. This event should always be reported as an error, as there is no way to determine which SEEN is the "correct" one.

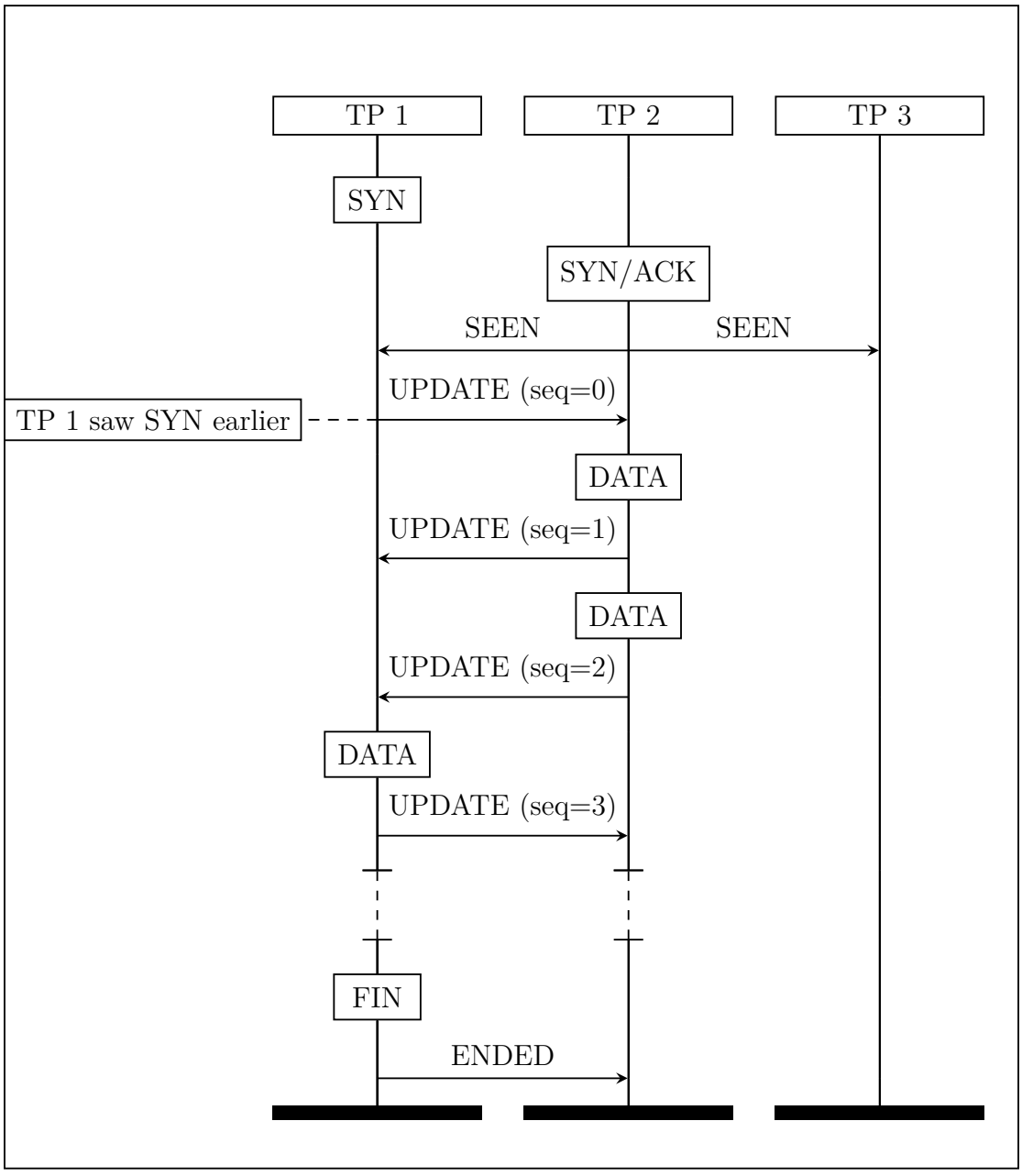

Figure 1.12: FlowSync with 3 TPs

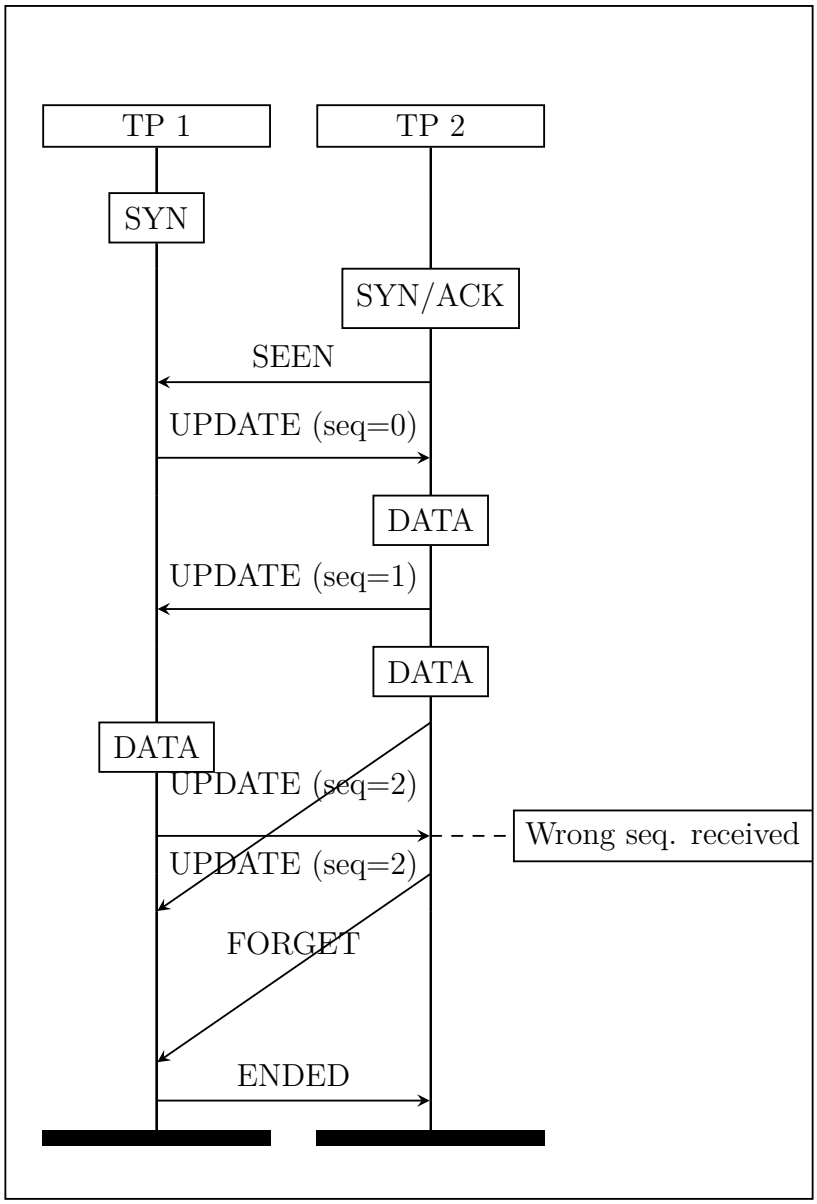

Figure 1.13: Out-of-Sync example

1.3.3 Statistics synchronization

Another consequence of asymmetric traffic is that flow statistics, such as the number of bytes and packets in each direction, are not contained in a single location, but instead split across two TPs.

External consumers of these statistics, such as a database or monitoring interface, typically need information about both directions. This creates a need to aggregate connection statistics per-flow, not just per-direction. One way to do this is to aggregate the per-direction data in an external database, to which each TP reports directly. However, this solution has poor performance when dealing with a very large number of flows, since the database must perform expensive lookup operations to match the upstream and downstream data points.

Another possible solution to the statistics reporting problem is aggregating statistics over the FlowSync domains. This would mean that a single TP reports all the statistics for a connection to the database. The advantages of this solution are that no external database nor extra hardware is required. However, this solution is susceptible to interruptions and delays on the FlowSync network, e.g., late packets and drops. Also, the additional data reported with each update packet adds bandwidth load to the network.

In the event of dropped statistics updates, different policies for handling this can be implemented. For example, re-send the update, or report statistics over a separate channel to the external database. In this scenario, it is necessary to ensure that duplicate accounting of statistics is not possible.

The statistics synchronization feature has undergone several iterations since it was first implemented at Sandvine. The initial implementation assumed that the FlowSync domains, which operate at the link layer, cannot drop packets. In reality, this is naturally not the case, and subsequent implementations have attempted to solve this issue by introducing acknowledgments for most packets. In total, there are 10 packet types in the current implementation:

1. **Connection update**: Sent from downstream to upstream TP. Contains the statistics information of the downstream side of the flow.
2. **Connection update acknowledgment**: Sent from upstream to downstream TP in response to connection update. Acknowledges the update and includes the current statistics of the upstream side.
3. **Request for acknowledgment**: Sent from downstream to upstream TP if acknowledgment is not received within a configurable timeout.
4. **FlowSync ENDED**: Same as used in classification, indicates that the connection has been closed.
5. **Reject**: Sent from either TP in response to any message other than Reject, indicating that synchronization of this connection's statistics has ended.
6. **Final**: The final statistics update. Sent from downstream to upstream TP.

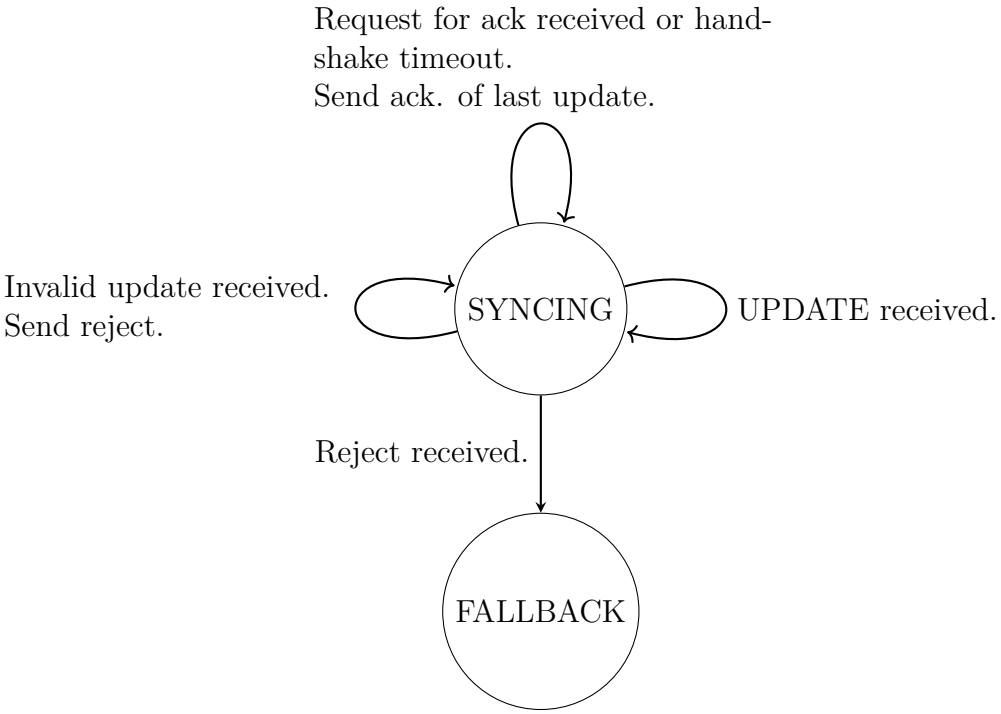

Figure 1.14: State diagram of an upstream TP performing statistics synchronization.

7. **Final acknowledgement**: Acknowledgement of the final statistics update. Sent from upstream to downstream TP.
8. **Request**: Requests the final statistics update from downstream TP. Sent from upstream to downstream TP.
9. **Response**: Final statistics update, sent in response to the Request message.
10. **Response acknowledgement**: Acknowledgement of the Response message.

The upstream TP is the one that observes the packets from the client to the server. Which TP is considered the upstream can be determined by observation of the SYN packet since that is only ever sent in the upstream direction. Figure 1.14 depicts a flow chart showing the states and actions of the upstream TP during synchronization. Receiving an UPDATE message updates the upstream TP's connection metadata with the statistics from the downstream TP. Receiving a request for an acknowledgment message causes the upstream TP to send an acknowledgment, including a receipt of the statistics received so far. When an invalid update is received, that is, one with an incorrect sequence number, a reject message is sent. Upon receiving a reject message, the TP enters the FALLBACK state and no further synchronization is performed. The upstream TP is also responsible for reporting the total statistics data to the external database. This happens periodically during synchronization if the statistics data have changed since the last time they were reported to the database.

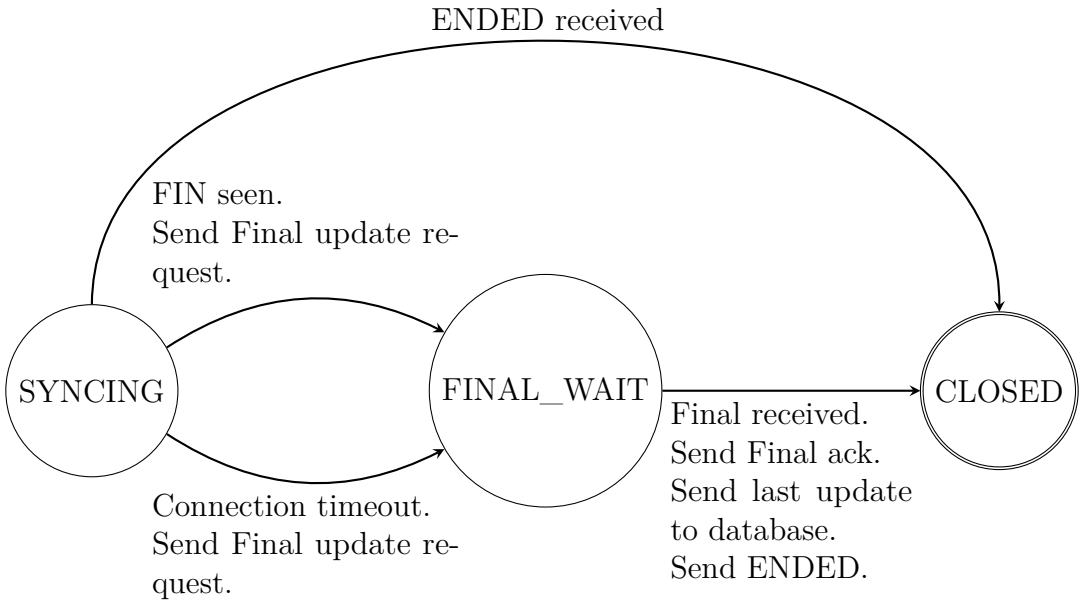

Figure 1.15: State diagram showing closing of statistics sync at the upstream TP.

Figure 1.15 shows how a connection closing is handled by the upstream TP. Receiving ENDED indicates that the connection was closed at the downstream TP and can be closed. When FIN is seen on the traffic channel or the connection times out, a request for the final statistics update is sent to the downstream TP, and the upstream TP enters the FINAL_WAIT state. Once the final update has been received, the last update can be sent to the external database, and ENDED is sent to the downstream TP.

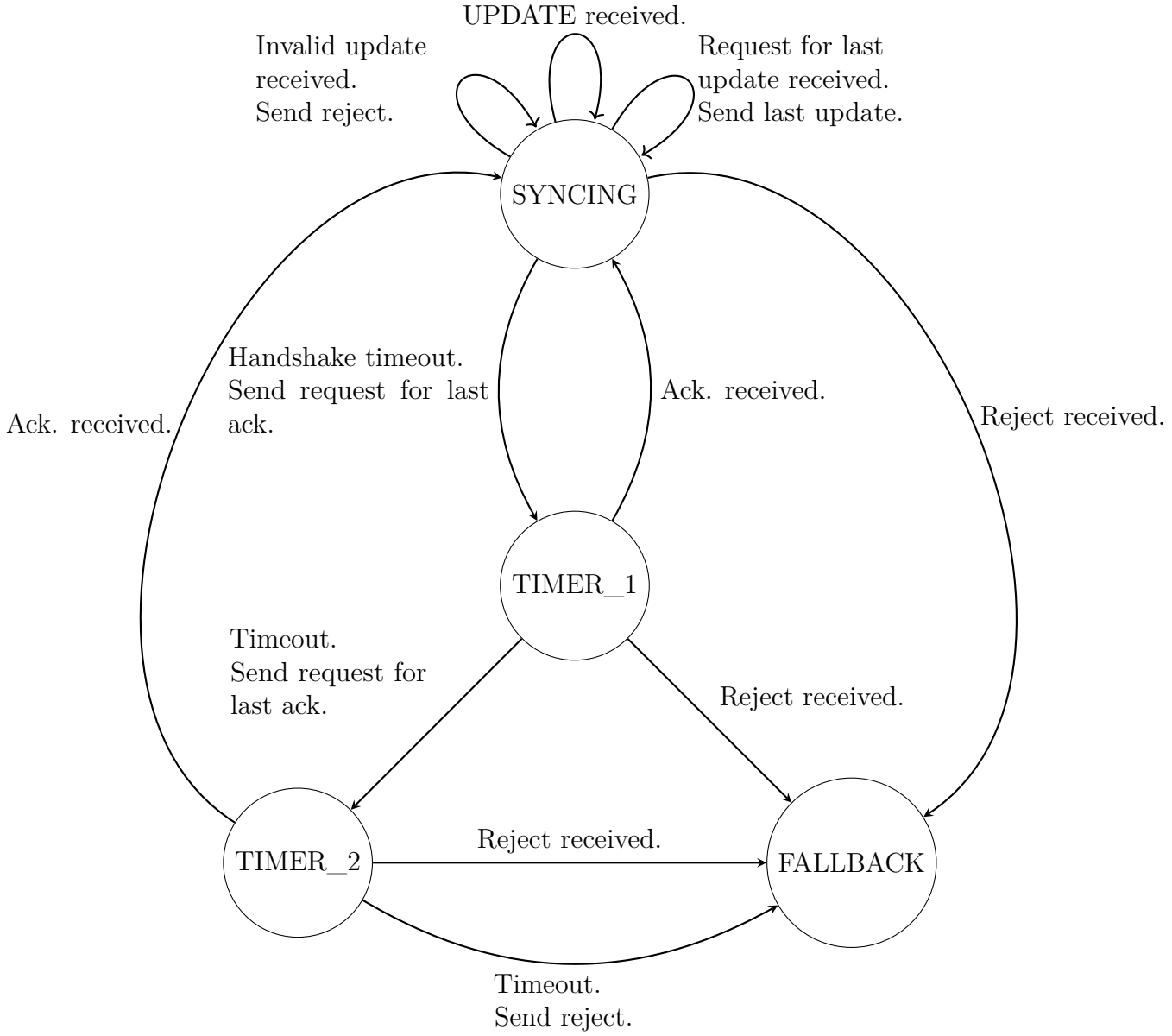

Figure 1.16: State diagram of a downstream TP performing statistics synchronization.

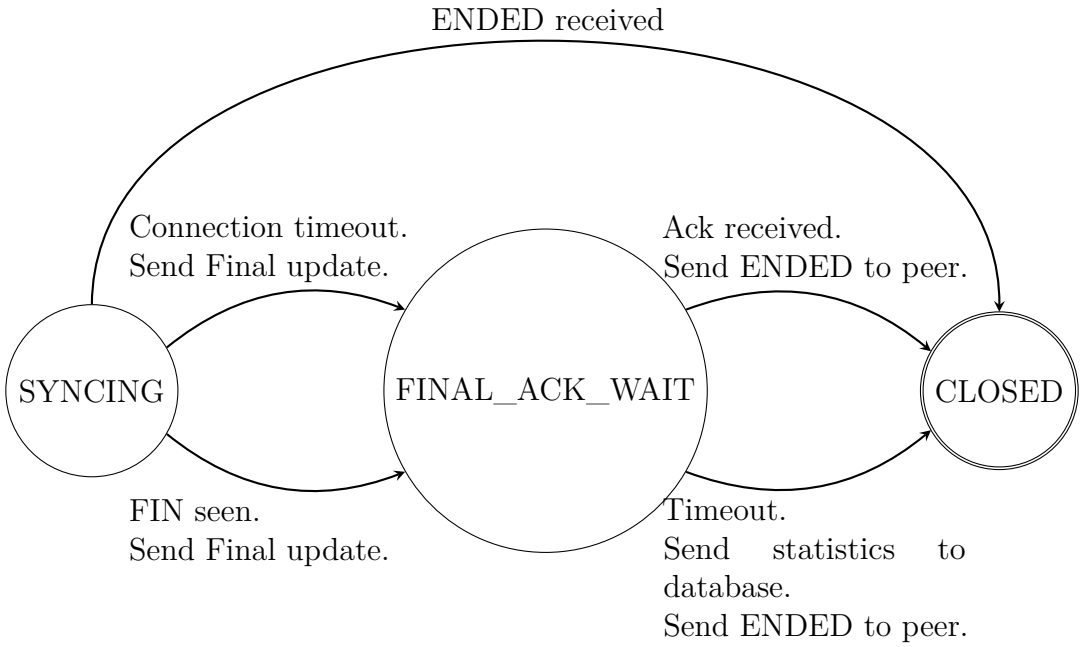

Figure 1.17: State diagram showing closing of statistics sync at the downstream TP.

The downstream TP is the one that sees the traffic from the server to the client, including the SYNACK packet, which is only seen in this direction. The behavior of the downstream TP is essentially a mirror of the one at the upstream, but with additional timers to allow for some drops and delays on the FlowSync network. The flowchart in Figure 1.16 shows an overview of the synchronization behavior. Instead of sending updates to the external database, the downstream TP sends statistics updates to the upstream TP. At any point, a request for acknowledgment can be sent to the upstream TP, and the downstream TP moves to the TIMER_1 state. If no acknowledgment is received before this next timeout, another request is sent, and the downstream TP moves to state TIMER_2. If this also times out, a reject message is sent, the FALLBACK state is entered, and synchronization is stopped. In the FALLBACK state, the downstream TP sends its statistics data directly to the external database instead of the upstream TP.

The procedure for reporting the final statistics when a connection is closed is shown in Figure 1.17. The main difference from the upstream closing procedure is that the final statistics data is sent directly to the external database if the final update acknowledgment is not received in time.

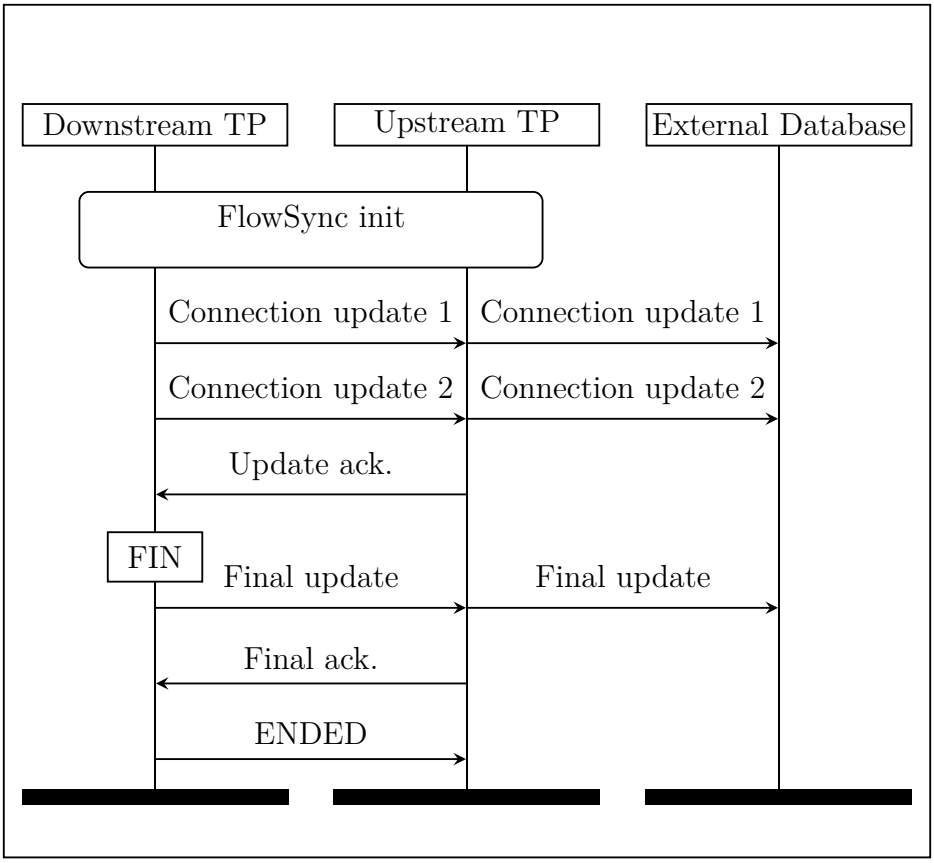

Figure 1.18: Message flow chart of a statistics synchronization example.

Figure 1.18 shows a subset of messages exchanged as part of a complete successful statistics synchronization conversation between two TPs. In this example, no packets are dropped between the upstream and downstream TP, so no re-sending of messages is necessary. The connection is ended with a FIN message seen by the downstream TP, which sends its final update to the upstream. After the acknowledgment of the final statistics update, the FlowSync conversation is ended with the ENDED message.

When in fallback mode, each TP sends its half of the connection statistics directly to the external database. The statistics sent in fallback mode are the statistics collected since the last successful synchronized update, as far as each TP can tell. For the upstream, that means the last update sent to the external database. For the downstream, it means the statistics last received in an update acknowledgment. The reason for this, according to the internal design notes, is to eliminate the possibility of duplicate statistics.

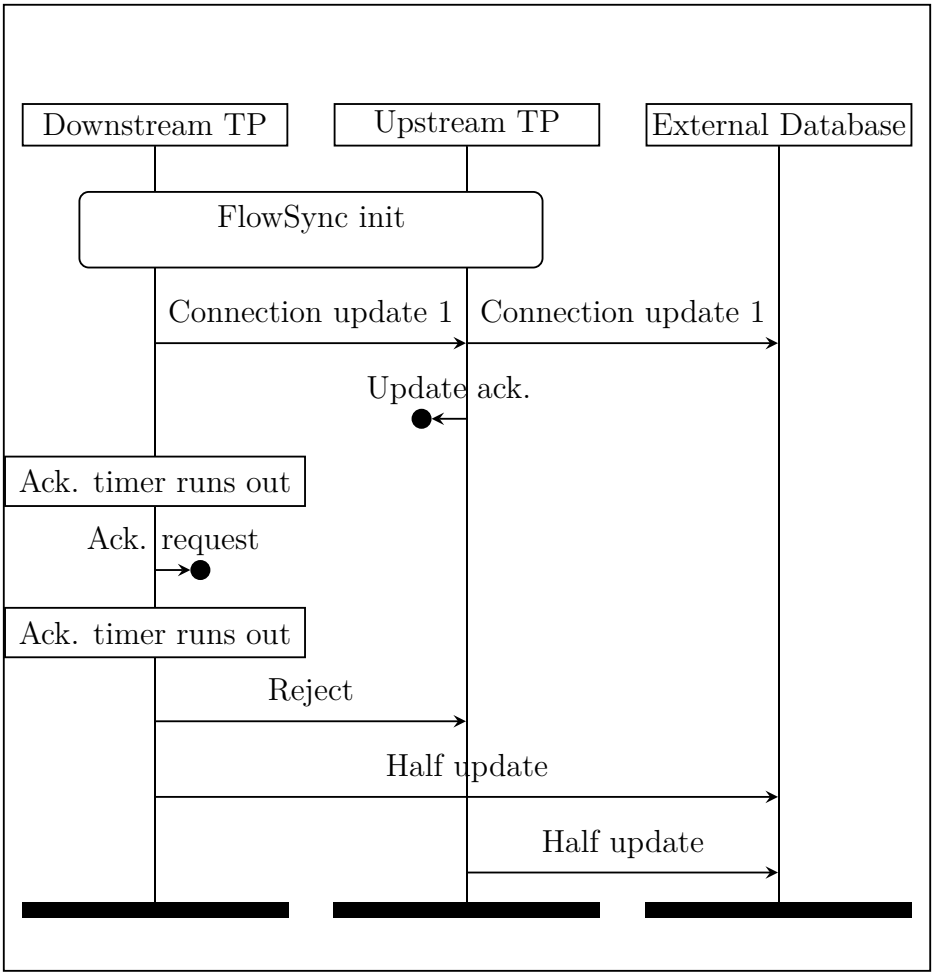

Figure 1.19: Statistics synchronization fallback example

Figure 1.19 shows an example of a statistics synchronization conversation that results in the fallback behavior. The downstream TP does not receive the update acknowledgment before its first timeout and sends another request for acknowledgment. The second update acknowledgment is also dropped, resulting in the downstream TP sending a reject message and entering the FALLBACK state (See Figure 1.16). From this point forward, the downstream TP sends half updates to the external database. Once the upstream TP has received the reject message, it also enters the FALLBACK state (See Figure 1.14) and starts sending half updates to the database.

2

Problem

This chapter defines the problem this thesis aims to solve and explains the limitations of the scope of the project.

A cluster of Traffic Processors sharing traffic metadata using FlowSync is an example of a distributed system. Distributed systems are notoriously difficult to reason about intuitively due to the inherently complex nature of concurrent systems in general. *Formal methods* can be employed to aid in the reasoning about concurrent systems [13]. By reasoning formally about the system and verifying its correctness using either a formal *proof* or *model checking*, faults in the protocol that might otherwise cause system failure can be identified before they manifest in the implementation.

The problem that this project intends to solve is that of accurately modeling the FlowSync protocol in such a way that formal methods can be applied in order to verify its correctness or identify its flaws.

The problem statement is summarized by the following research questions:

1. How can a large system for internet traffic management be compactly modeled in a formal language while still being amenable to verification?
2. What properties can be verified, and what issues can be found by verifying the model?
3. How can the results of verification be used to correct or limit the impact of any issues found?

In order to answer these questions, FlowSync is modeled, and properties corresponding to the desired behaviors outlined in Section 1.3 are verified. This serves as a case study of how formal methods can be applied to find the root causes of issues in a design, as well as how the results can be used to fix or limit the impact of said issues.

As stated in the introduction, the classification may be used to filter out harmful traffic, such as illegal content or malware. Because of this, there is an incentive for malicious users to try to bypass the classification system. Thus, another aspect of FlowSync to be verified by the model is whether it is possible to craft a traffic flow such that classification fails due to some protocol flaw or by overloading the FlowSync domain(s).

This project's scope is limited to the FlowSync protocol for enabling classification and statistics collection for traffic flows. Complex policies applied to each type of traffic, such as volume-based shaping and traffic redirection, will not be covered. Instead, a simple class-based filtering policy is modeled, where traffic is forwarded or dropped based on the state of the classifier. Statistics reporting is simplified to only mean packet count, and the external database is assumed to only aggregate the reports as the total number of packets in each flow direction.

Only TCP will be considered as the transport layer protocol. The case for User Datagram Protocol (UDP) is slightly more complicated, as there are no rules for how a UDP conversation starts or ends. This means that, for example, peers may start streaming packets over UDP after negotiating in an otherwise unrelated TCP stream, as is the case for e.g. VoIP applications. In this case, the ordinary SEEN-UPDATE handshake cannot be used, and the conversation must be classified by some other means.

3

Theory

This chapter covers existing theory and related work on the verification of distributed systems in general, verification of traffic management systems, model checkers like TLA$^+$, R-CHECK, and SPIN for modeling distributed systems, and an alternative approach to formal verification of large protocol implementations.

3.1 Formal methods and model checking

As computer systems have become more complex and are increasingly used in critical applications, such as in vehicles and medical equipment, where the potential risks include financial consequences or even loss of life, the need to gain confidence in the correctness of such systems has increased. This section covers *formal methods* and, more specifically, *model checking*, which is a powerful, automated verification technique that can provide strong evidence for the correctness of a system. Generally, formal verification consists of two steps; specification and verification. Specification involves specifying the behavior of the system (referred to as the *model* in the context of model checking), as well as intended properties, in a formal language. Verification consists of proving, either by manually constructing a proof or by automated checking, that the system description satisfies the properties [13].

A number of formal methods paradigms, and techniques have been developed for different use cases. Each technique offers unique capabilities for verifying different aspects of software and hardware systems. The following is a summary of a selection of techniques and how they can be applied:

1. **Symbolic execution** is a technique that explores the possible execution paths of a program by using symbolic values instead of concrete inputs. This approach is helpful while analyzing and verifying cryptographic protocols and low-level system components [14].

2. **Model checkers** are used to verify that a given model of a system satisfies specific properties, typically expressed in a temporal logic like Linear Temporal Logic (LTL). Model checkers like SPIN, NuSMV, and UPPAAL can be applied to verify concurrent and distributed systems, real-time systems, and communication protocols, among other applications [15].

In [16], the authors describe how model checking emerged as an alternative to manual proof-based formal methods in response to manual proofs of larger systems becoming impractical. Model checking typically involves describing the system in terms of its states and the transitions between those states. The specification of the system is then written as a formula in some variant of temporal logic. The process of verifying whether the system description is a model of a temporal logic formula gives the term model checking.

One of the strengths of model checking is that the verification is automatic and that it provides counterexamples if the model does not satisfy them, whereas its main weakness is the *state explosion problem* [13]. The state explosion problem refers to the fact that the number of possible system states increases exponentially in the number of variables or agents in the system [17].

3. **Sound static analyzers** examine the source code of a program without actually executing it, ensuring that the analysis is both sound (no false negatives) and complete (no false positives). Tools like Frama-C and eThor can verify various software systems, including device drivers, embedded systems, and safety-critical applications such as aviation software and the Secure Sockets Layer (SSL) stack [18], [19].

4. **Verified runtime monitoring** involves using formal methods to verify the correctness of a runtime monitor, which in turn checks the behavior of a running system against a specified set of properties. This approach can be applied to various systems, including web applications (browser sandbox), distributed systems, and safety-critical systems. Logical Automata for Runtime Verification and Analysis(LARVA) and JavaMOP support verified runtime monitoring [20], [21].

3.2 Temporal logic

The temporal logic of programs [22] refers to the time-related properties of programs. The two main categories of such properties are *invariance* properties and *eventuality* properties. Invariance properties are properties that hold globally for all states of a system. Eventuality properties express that given some state, a state where the property holds must be reached in a finite amount of time. These two categories correspond to Lamport's [23] notion of *safety* and *liveness* properties.

Linear Temporal Logic (LTL) refers to logic extended with quantifiers over all paths of execution in a state transition system, thus specializing the general notion of time to refer strictly to such paths. Common quantifiers include

- $\Box\ \phi$ ("box", "always"), which states that a formula, ϕ, always holds from this point forward,
- $\Diamond\ \phi$ ("diamond", "eventually"), which states that a formula, ϕ, will hold at some point in the future, and
- $\phi\ \mathcal{U}\ \psi$ ("until"), which states that "ϕ holds until ψ holds".

The "box" and "diamond" quantifiers can be combined to express more complex properties such as the following:

- $\Diamond \Box \phi$ - Stability: After a finite amount of time, ϕ holds and continues to hold indefinitely.
- $\Box \Diamond \phi$ - Recurrence: ϕ holds infinitely often.
- $\Box (\phi \implies \Diamond \psi)$ - Response: An action ϕ must be followed by a response ψ within a finite amount of time.

For example, consider an elevator that can move up and down to 3 different floors. A *safety* property, such as "the elevator may not move while the doors are open", can be specified in LTL as follows: $\Box$ (*door is open* $\Rightarrow$ *elevator is stationary*). *Liveness* properties can also be specified, for example, "whenever the button for floor 3 is pressed, the elevator eventually reaches floor 3", can be expressed as $\Box$ (*button 3 pressed* $\Rightarrow \Diamond$*elevator is on floor 3*).

3.3 Compositional model checking

As previously stated, the state explosion problem is one of the main drawbacks of the model checking approach to formal verification. *Compositional model checking* refers to methods for overcoming the state space explosion problem by breaking the model into smaller components. These component models can then be checked separately, thus reducing the size of the state space to be explored for each component. One such method is called *assume-guarantee reasoning*, which involves defining guarantees provided by each component, given some assumptions about the global state of other components in the system [24].

The notation used for assume-guarantee reasoning is $\langle\phi\rangle P \langle\psi\rangle$. This expression is true when the process P guarantees the property ψ under the guarantee ϕ. We can chain these assume-guarantee expressions to perform a deduction as seen in Equation 3.1, which reads "Given that P_1 guarantees ϕ and that P_2 guarantees ψ under the assumption ϕ, it can be deduced that the parallel composition of P_1 and P_2 guarantees ψ".

$$\frac{\langle true\rangle P_1 \langle\phi\rangle \qquad \langle\phi\rangle P_2 \langle\psi\rangle}{\langle true\rangle P_1 \| P_2 \langle\psi\rangle} \tag{3.1}$$

3.4 Verification of network management protocols

Software Defined Networking (SDN) is an approach to network management in which the network is separated into a control-plane and a data-plane. The control-plane is responsible for orchestrating the network and providing a centralized point of configuration. In the data-plane, traffic is forwarded, dropped, shaped, or otherwise handled according to the configuration determined by the control-plane.

Previous research [25] explores the possibility of building a verified network controller based on the OpenFlow protocol for SDNs. The approach involves creating several layers of abstraction, from a high-level network specification language called NetCore, down to a formal model of the switches used to implement the network. Each translation from higher to lower is proven correct with regards to the semantics of each using the Coq proof assistant. Using this approach, the authors show that invariants such as "all nodes are reachable" can be proven to hold for a given control-plane configuration.

The TPs defined earlier in this thesis are an example of forwarding nodes in an SDN architecture. However, FlowSync does not interact directly with the control-plane, nor does it play a part in configuring the network according to a specification. Rather, it facilitates sharing of metadata between nodes in the data-plane. Nevertheless, this previous research shows the feasibility of applying formal methods to verify an SDN system.

3.5 Running protocol implementations in a modeled environment

A different approach to model checking large protocol implementations is described in [26]. In their paper, the authors run the actual implementation of the Linux TCP stack in CMC, a C model checker [27]. The benefit of running the actual implementation in a modeled environment is that there is no need to model the protocol separately from the implementation. Thus, this approach does not suffer from the issue of the model not matching the implementation, potentially hiding issues. The main drawback is the amount of work required to model the environment that the implementation runs in.

This approach could be used to model FlowSync, given enough time and effort to lift the implementation of the whole traffic processing system into a modeled environment. However, given the time constraints of this project, this approach is deemed infeasible.

3.6 Modeling with Promela and SPIN

Promela (Process/Protocol Meta Language) is a modeling language designed primarily for specifying concurrent and distributed systems, protocols, and algorithms. It was developed by Gerard J. Holzmann to make it easier to model complicated systems in a clear and accessible way. It helps designers capture the dynamic behavior of systems and identify errors, inconsistencies, or deadlocks before the actual implementation [28].

SPIN is a software verification tool designed to examine and confirm the accuracy of Promela-modeled systems. SPIN can verify several properties, including an absence of deadlocks (ensuring that no process is forever delayed), liveness (ensuring that good states eventually occur), and safety (the absence of bad states) [29]. Fairness is needed here to ensure all possible actions have a fair opportunity to occur. Actions correspond to a step or an event that triggers the state change or transition within the process or model. It can include activities like sending or receiving messages or performing an operation. Once the preconditions for the action are satisfied, then that action is enabled and may be executed in the next step.

When verifying liveness properties, the fairness of the model must be considered. For example, consider a model of two processes that both count the integers 1, 2, 3, ..., and so on. A liveness property may state that both processes eventually reach 100. However, if the model is not fair, then it is possible that only one of the processes gets executed and the other never reaches 100, which violates the property.

There are two categories of fairness: weak fairness and strong fairness. Weak fairness implies that if an action is always enabled, it will eventually be executed. In contrast, strong fairness guarantees that if a process is enabled infinitely often, it will eventually be executed [30]. Note the subtle difference between *always* and *infinitely often*. For an action to be considered as always enabled along an execution path, it must be enabled in every step along this path. An action that is only enabled every other step, or every $n \in \mathbb{N}$ steps, is still considered enabled infinitely often. SPIN supports weak fairness by default [31].

Buffered channels are a fundamental feature of Promela that facilitates communication between processes. Channels can be declared with a fixed buffer size, and they implement the message passing through the send (!) and receive (?) operators. *Send* is a non-blocking operation when the channel is not full. Promela also supports a "rendezvous channel" with a buffer size of 0. Sending a message on a rendezvous channel blocks the sender until the receiver is available and can thus be used for synchronization. *Receive* operation is blocking as it has to wait until a matching message appears in the channel. The channel *Poll* operation can be used to check whether receive operation can be performed or not. This provides a natural way to model synchronization and inter-process communication in distributed systems [32].

3.7 Modeling distributed systems in TLA$^+$

TLA$^+$ is a language for specifying systems as mathematical formulas [33]. A TLA$^+$ specification usually describes the system by its states and the allowed transitions (behaviors) between these states. The system builds upon the Temporal Logic of Actions (TLA), which in turn builds upon temporal logic. The TLA$^+$ Toolkit also provides a high-level language called PlusCal, which allows the modeling of processes in a procedural language with a syntax similar to Pascal.

TLA$^+$ has previously been used to verify both the Pastry [34] and Chord [35] protocols for distributed hash tables. In the case of a distributed data structure, the main challenge is ensuring that all nodes involved have a shared understanding of the structure's state, even in the event of nodes joining or leaving the network without prior notice. The main fault discovered in the paper by Lu et. al. [34] was in the case where two nodes responsible for overlapping sections of the hash table joined simultaneously. In this case, the nodes would be aware of the existing members of the network but not each other. The authors also proposed a fix for the protocol to prevent this issue.

In contrast to [34], which takes a high-level approach to modeling the network nodes, it will be necessary to model low-level components of the system, including link-level addressing and routing. This is because FlowSync includes mechanisms for failover in the event of a single direction of a single link becoming unavailable, which should be included in the model.

3.8 Modeling multi-agent systems in R-CHECK

Multi-Agent Systems (MAS) emerged as new software technologies in the last decade that offer an efficient and more natural alternative to building intelligent systems, thereby providing a solution to the current complicated real-world problems. MAS consists of agents that work together either to accomplish the same or discordant tasks. Agents can communicate with broadcast, unicast, or a combination of both. Modeling a multi-agent system that reconfigures the way agents behave depending on a state change is being researched heavily.

R-CHECK is a model-checking toolkit for verifying and simulating reconfigurable multi-agent systems [36], [37]. It is built on top of *nuXmv*, which is a symbolic model checker for analyzing synchronous finite-state and infinite-state systems [38]. This enables the toolkit to perform LTL symbolic model checking. A modeling convenience for interaction aspects like reconfiguration, coalition building, and self-organization is provided by R-CHECK, which also allows a high-level input language with symbolic semantics. Users can give input to the interactive simulator to tell how to simulate the system or let the simulator perform an arbitrary execution.

4

Method

This chapter outlines the approach used to model and verify FlowSync in order to answer the research questions.

The approach is divided into two main sequential steps. First, a simplified implementation of the classifier sharing algorithm was modeled in three separate modeling frameworks: R-CHECK, TLA$^+$, and SPIN. This was done to gauge the suitability of each framework for modeling FlowSync, since they each have a unique set of mechanisms for expressing and verifying behaviors. The second step consisted of modeling the remaining behaviors of FlowSync; statistics synchronization, and link redundancy, in one of the three frameworks. The choice of the framework was based on a suitability judgment guided by the results of the first step. See Section 5.3 for the reasoning supporting the choice of modeling framework.

As noted by [17], one of the concerns when modeling a system is managing the complexity of the model. The model must be detailed enough to capture the behaviors to be verified while being compact enough to be verifiable in a reasonable amount of time. In order to reduce the complexity of the model, the following simplifications were made:

- Link over-capacity was not considered. By modeling the receive queues of all agents to block the sender when they are full, there is an implicit assumption that the sender will never send at a higher rate than the receiver can handle. This simplification has the disadvantage of not allowing modeling of Denial of Service (DoS) attacks but still allows for verification of other behaviors.
- Only stable link states were modeled. Since there is no mechanism in the link redundancy layer of FlowSync to detect link failure other than heartbeat timeout, packet drops are inevitable in the time span between a link going down and this being detected by the TPs on the network.
- Link redundancy was modeled separately from classifier sharing and statistics synchronization. This separation relies on assume-guarantee reasoning as described in Section 3.3. The link redundancy model guarantees that if a message is sent to a TP that is available on the FlowSync network, that message will arrive at its destination. This guarantee was then used as an assumption in the model of classifier sharing and statistics synchronization.

- TPs were modeled as unidirectional, meaning that each TP either only forwards packets upstream or downstream. In the real world, a single TP may handle both upstream and downstream traffic.
- Only a single TCP connection was modeled. This was motivated by the fact that, for the purposes of classification and statistics data collection, connections do not interact with each other. Concurrent updates to a single connection's metadata are prevented by locking the associated row of the TP's internal connection table during the handling of each packet belonging to that connection.

5

Survey of modeling frameworks

In order to get an overview of the modeling frameworks available and to determine which is most suitable for modeling FlowSync, a model of the classifier sharing protocol is implemented in three different frameworks. This model is further simplified in the following ways: (1) The classifier state is not considered; FlowSync updates are sent for every traffic packet observed. (2) The model is terminating, meaning that the model reaches a final state once the TCP connection between the client and server has ended.

5.1 Modeling

The main property of the behavior to be verified, apart from the absence of deadlocks, is that "given that the RTT on the FlowSync network is shorter than that of the traffic peers (server and client) and that no FlowSync messages are lost or reordered, the FlowSync peers share the classifier state for the connection until it ends". Expressing this informal requirement formally, in the form of an LTL formula, requires some simplification. Reasoning about RTT, or any other time, in LTL is inconvenient due to the fact that adding a timestamp to each state makes otherwise identical states unique, thus increasing the size of the state space. Instead of using absolute time, the relative time property can be expressed formally as: "If there are FlowSync messages in the receive queue of a TP, they will be handled before any other traffic".[1] In summary, the desired property of FlowSync classifier sharing can be expressed with the formula shown in equation 5.1.

$$\Box\,(\forall\, tp \in TP : \textit{tp's FlowSync queue is not empty} \implies \textit{tp does not handle traffic}) \implies \Box\,(\neg\ \textit{out of sync}) \quad (5.1)$$

In order to verify that this property is not being vacuously satisfied by the model entering a deadlock state before out-of-sync can occur, a property is added stating that the client eventually completes its conversation with the server and reaches its final state. The LTL formula for this is shown in equation 5.2

[1] Note that the FlowSync priority requirement is actually stricter than the RTT requirement. Only requiring a shorter RTT does not protect FlowSync from bursts of packets in both directions at the same time, which may cause out-of-sync.

$$\Diamond\ (Client.state = DONE) \tag{5.2}$$

TPs going out of sync is detected using sequence numbers as described in Section 1.3, and the absence of lost FlowSync messages and reordering is modeled using First In, First Out (FIFO) receive queues for each TP.

The results of the survey are described in Section 5.2 and discussed in Section 5.3.

5.1.1 R-CHECK

R-CHECK is a language for modeling Multi-Agent Systems that communicate synchronously. The agent concept is similar to the process used in SPIN and PlusCal, with each agent having its own state and communicating with other agents over message channels.

$$system = Engine(engine1, \ldots) \parallel Engine(engine2, \ldots) \parallel Server(server, \ldots) \parallel Client(client, \ldots) \tag{5.3}$$

Equation 5.3 shows how a system consisting of two TPs, a server, and a client is defined by the parallel composition of agents. The initialization of each agent's communication channels is left out for brevity. Appendix B contains the complete system and agent definitions.

```
1 SPEC F (engine1-syncing && engine2-syncing)
2 SPEC F (server-tstate == closed && client-tstate == closed)
3 SPEC G (!engine1-outofsync && !engine2-outofsync)
4
```

Figure 5.1: Safety and liveness properties expressed as R-CHECK LTL formulas.

Figure 5.1 shows the LTL formulas for each of the properties to be verified. These correspond to the formulas in Figure 5.5, specifying that "TPs eventually sync", "the TCP connection eventually finishes" and "TPs never go out of sync".

5.1.2 TLA$^+$

TLA$^+$ itself is a mathematical language used for describing systems using formulas [33]. Thus, the concept of processes does not directly apply to TLA$^+$. Instead, the actions of each process are modeled as actions on the global state, which means that care must be taken to only access and update the state of the part of the global state that belongs to the process whose action is being taken. The high-level PlusCal language makes this easier, as it allows for defining processes, each with their own local state. This language is used to define the component processes of the system, and is then compiled to TLA$^+$ for verification in TLC, the TLA$^+$ model checker.

```
macro Send(name, msg) begin
    channels[name] := @ (+) msg;
end macro;

macro Receive(name, msg) begin
    await channels[name] /= <<>>;
    msg := Head(channels[name]);
    channels[name] := Tail(channels[name]);
end macro;

```

Figure 5.2: Definition of the `Send` and `Receive` macros in PlusCal.

$$WillSync \triangleq \Diamond \forall\, tp \in TPs : state[tp] = \text{“SYNCING”}$$

$$NeverDesync \triangleq \Box \forall\, tp \in TPs : outofsync[tp] = \text{FALSE}$$

$$FlowsyncPriority \triangleq \Box \forall\, tp \in TPs : (flowsync\_channels[flowin[tp]] \neq \langle\rangle) \Rightarrow (pc[tp] \notin \{\text{“TP\_traffic\_handle”}, \text{“TP\_traffic\_forward”}\})$$

$$WillSyncWithFlowsyncPriority \triangleq FlowsyncPriority \Rightarrow WillSync$$

$$NeverDesyncWithFlowsyncPriority \triangleq FlowsyncPriority \Rightarrow NeverDesync$$

Figure 5.3: Safety and liveness properties expressed as TLA$^+$ formulas.

TLA$^+$ does not support message passing or channels natively. Instead, message passing channels are implemented using the built-in *Sequences* module. Using sequences the `Send` and `Receive` macros are defined as shown in Figure 5.2 to allow message passing similar to the `!` and `?` commands in Promela. The `Send` macro appends the message to the end of the channel sequence using the concatenation operator `(+)`. The `Receive` macro blocks until the channel sequence is non-empty, and removes its first element using the `Head` and `Tail` functions. Using these macros allows the code to be structured similarly to that of the R-CHECK and Promela versions, which facilitates a more direct comparison of the models. See Appendix C for the complete model of simplified classifier sharing in PlusCal.

The desired safety and liveness properties of FlowSync are expressed as plain TLA$^+$ formulas, as shown in Figure 5.3. Note the use of the forall operator ($\forall$) to quantify over *all* TPs in the model, meaning that this model can easily be extended to include more TPs without changing the formulas.

```
init {
    run TP(FlowSync[0], FlowSync[1], Traffic[0], Traffic[1]);
    run TP(FlowSync[1], FlowSync[0], Traffic[3], Traffic[2]);

    run Server(Traffic[2], Traffic[0]);
    run Client(Traffic[1], Traffic[3]);
}

```

Figure 5.4: System definition in Promela, see appendix A for the complete model.

```
#define FLOWSYNC_PRIO(i) (TP[i]@handle_traffic -> (len(FlowSync[i -
    1]) == 0))
#define FLOWSYNC_PRIO_ALL ([] (FLOWSYNC_PRIO(1) && FLOWSYNC_PRIO(2)
    ))

ltl eventually_sync {
    FLOWSYNC_PRIO_ALL -> (<> (TP[1]:state == FS_SYNCING && TP[2]:
    state == FS_SYNCING))
}
ltl eventually_fin {
    FLOWSYNC_PRIO_ALL -> (<> Client@done && <> Server@done)
}
ltl never_desync {
    FLOWSYNC_PRIO_ALL -> ([] ! (TP[1]:out_of_sync || TP[2]:
    out_of_sync))
}

```

Figure 5.5: Safety and liveness properties expressed as SPIN LTL formulas.

5.1.3 SPIN

SPIN has built-in support for processes defined using the `proctype` keyword, which enables modeling of TPs, a TCP server and a TCP client as separate processes. Figure 5.4 shows how the system is defined as the parallel composition of these processes by running them from the `init` process. Promela also supports asynchronous message passing using buffered channels, which operate similarly to the receive queues used in a hardware Network Interface Card (NIC).

Figure 5.5 shows the LTL formulas corresponding to the behaviors to be verified. Macro definitions are used to avoid duplication of the common `FLOWSYNC_PRIO_ALL` condition, which states that FlowSync packets are always handled before traffic packets. The first formula, `eventually_sync`, is a liveness condition that verifies that both TPs eventually reach the SYNCING state. The second, `eventually_fin`, states that the `Client` and `Server` processes eventually reach their final states, indicating that the TCP connection finishes successfully. The third, `never_desync`, is a safety property, stating that the TPs never reach an out-of-sync state.

Property	FlowsyncPriority	Time (s)	Errors
WillSync	Yes	70	0
WillFinish	Yes	81	0
NeverDesync	Yes	87	0
WillSync	No	85	0
WillFinish	No	166	0
NeverDesync	No	1	1

Table 5.1: Results of model checking the simplified model of classifier sharing in TLA$^+$.

Property	`FLOWSYNC_PRIO_ALL`	Time (s)	Errors
`eventually_sync`	Yes	0.00	0
`eventually_fin`	Yes	1.74	0
`never_desync`	Yes	0.12	0
`eventually_sync`	No	0.00	1
`eventually_fin`	No	515	0
`never_desync`	No	0.01	1

Table 5.2: Results of model checking the simplified model of classifier sharing in SPIN.

5.2 Verification

This section presents the results of verifying each of the models presented in the previous section in their respective model checkers.

Checking the R-CHECK model produces an error when verifying the "eventually finish" property. This is due to the absence of buffering of the message channels used for sending FlowSync messages, which makes it possible for this model to deadlock with both TPs trying to send messages to each other at the same time.

Table 5.1 shows the results of model checking the TLA$^+$ model of simple classifier sharing in the TLC model checker. The NeverDesync property fails to verify when FlowSync priority is disabled, since the FlowSync RTT requirement described in Section 5.1 is not enforced.

See Table 5.2 for the results of model checking the Promela model of the simplified classifier sharing service. Just like in the TLA$^+$ model, removing the precondition `FLOWSYNC_PRIO_ALL` causes verification to fail for the `eventually_sync` and `never_desync` properties.

5.3 Discussion

Modeling the same system in three separate frameworks allowed for a comparison of their suitability for this task. Some of the observations made were as follows:

1. The fact that channel communication in R-CHECK is synchronous poses a restriction on its ability to model an Ethernet network since there are no means of buffering messages in a receive queue. This can cause deadlocks in the model when an agent is waiting to send a message on a channel, which is not possible when using a real Ethernet implementation.
2. SPIN has native support for buffered message channels, whereas TLA$^+$ only supports sequences and unordered sets, and R-CHECK only supports synchronous (rendezvous) message passing. Buffered channels can be implemented in TLA$^+$ with relative ease, as shown in Section 5.2. However, no method of doing so in R-CHECK was found due to the lack of any sequence, list, or array data type.
3. R-CHECK and TLA$^+$ support quantifiers such as $\forall$ and $\exists$, which SPIN does not. This allows for easier scaling of LTL formulas, specifying system properties without duplicating code.
4. PlusCal does not allow multiple assignments to the same variable in a single labeled section since each such section becomes a single step in the TLA$^+$ translation. When a single global map is used to contain all channels, this puts a restriction on how large each atomic step can be. Each step may only send or receive a single message from a channel. This increases the number of states compared to the SPIN model, which allows an arbitrary number of send and receive operations in a single atomic execution step. This limitation can likely be avoided by foregoing PlusCal and using TLA$^+$ syntax to encode the model directly, computing the new value of each channel without using the `Send` and `Receive` macros defined in Section 5.1.2.
5. The performance of the SPIN model checker is significantly more than an order of magnitude, better than that of TLC for the behaviors and properties modeled.

Based on these observations, the choice was made to model the remaining parts of FlowSync in Promela, primarily due to its native support for asynchronous (buffered) message channels and its superior performance.

6

FlowSync link redundancy

This chapter covers the modeling and verification of the model of FlowSync link redundancy. The model is presented in Section 6.1. The results of verifying the model are presented in Section 6.2 and discussed in Section 6.3.

6.1 Modeling link redundancy

As mentioned in Section 1.3.1, a domain is only used if all the TPs in that domain can send and receive the packets. To verify this, a system similar to the one in Figure 1.9 is modeled, featuring three TP processes and two domains.

The link redundancy model can be seen as a combination of internal working of link redundancy and the high-level view of a service that uses the network for communication. That is, TPs periodically send heartbeat messages to check the active TPs in the domain, and at the same time, the services use the network for sending messages.

In real-world networks, arbitrary link failures can occur at any time. However, allowing unrestricted link failures within the model means including states where all or multiple links are down indefinitely, rendering all domains inactive. This necessitates a restriction of the model to examine the network's behavior under stable link status conditions, meaning that the status of a link is static for the duration of the heartbeat exchange.

The model includes two configurable constants, `N_LINK_FAILURE` and `THREAD_0_PRIO`, representing the number of link failures in the system and priority for the TP (`Thread0`) process. The name `Thread0` is borrowed from the real implementation, where heartbeat are sent and received on the main (first) thread of the system process.

```
#define N_TP 3
#define N_DOMAIN 2
#define N_LINK_FAILURE 2
#define THREAD_0_PRIO 2

init {
    // Set up links
    int i, j, k;
    for (i : 0 .. N_DOMAIN-1) {
        for (j : 0 .. N_TP-1) {
            links[i].tp[j] = true;
        }
    }

    // Disable N_LINK_FAILURE links
    for (k : 0 .. N_LINK_FAILURE-1) {
        select (i : 0 .. N_DOMAIN-1);
        select (j : 0 .. N_TP-1);
        links[i].tp[j] = false;
    }

    run Switch(switchs[0].domainMsgsChan, 0) priority 3;
    run Switch(switchs[1].domainMsgsChan, 1) priority 3;

    run Thread0(TPIn[0], 0) priority THREAD_0_PRIO;
    run Thread0(TPIn[1], 1) priority THREAD_0_PRIO;
    run Thread0(TPIn[2], 2) priority THREAD_0_PRIO;

    run Service(0, 1) priority 1;
}

```

Figure 6.1: Process priorities and initialization of link failure(s).

Following the defined number of link failures, the model randomly designates certain links as down by setting their statuses to false. This approach enables an exhaustive exploration of the system's behavior under various network disruptions. It is worth noting that the model also includes process priorities, which significantly restrict the system's behavior and response to these disruptions. The priorities are such that link redundancy messages are processed before any service messages. Essentially, this guarantees that before the service sends a service message, a stable network environment is already established. The process priorities in Promela are integer values, with a higher number representing a higher priority.

Figure 6.1 shows how the initialization of the model is done, along with different priorities of the processes and the use of constants. The complete model is shown in Appendix E.

Out of three processes, `Switch`, `Thread0`, and `Service`, the first two provide access to the FlowSync network, while the `Service` is using the network to send messages. The switch in this model is given the highest priority, designated as 3, owing to the necessity for the Switch to be fully operational before any TP commences message transmission. Following that, the `Thread0` process is given its priority according to the `THREAD_0_PRIO` constant. The rationale behind giving `Thread0` a higher priority than the `Service` is to ensure that the switch and TPs are prepared and stable before attempting to send any service messages. The `Service` process is assigned the lowest priority, marked as 1 since the service message should only be conveyed when the network layout has been established by the sending of heartbeat packets.

The purpose of the model is to verify that any service message sent on an active domain reaches its destination. The message delivery property *eventually_service* states that "Always when a message is sent by a Service, then eventually it will be received by the intended TP(Thread_0) process." The other property *domain_agreement* checks that "Eventually all the TPs will agree on the status of each domain." Figure 6.2 shows how the properties are expressed in SPIN LTL formulas.

```
ltl eventually_service {
    [] (Service[6]@sent -> <> Thread0[3 + 1]@receive_service)
}

ltl domain_agreement {
    <> [] (TPList[0].domain[0].ok == TPList[1].domain[0].ok
        && TPList[1].domain[0].ok == TPList[2].domain[0].ok
        && TPList[1].domain[1].ok == TPList[2].domain[1].ok
        && TPList[0].domain[1].ok == TPList[1].domain[1].ok
        )
}

```

Figure 6.2: Properties of Link Redundancy expressed as SPIN LTL formulas.

6.2 Verification

The link redundancy model aims to enable verification of the liveness property that packets sent by the service layer arrive at their destination. Table 6.1 shows the results of the verification of this property with various model parameters and Figure 6.3 shows the link failures corresponding to the failing `eventually_service` property on row 2 of the table. The complete trail for this run can be found in Appendix F.

Property	N_LINK_FAILURE	THREAD_0_PRIO	States	Time(s)	Errors
eventually_service	1	2	19,859	0.06	0
eventually_service	2	2	5,973	0.02	1
eventually_service	1	1	107	0	1
domain_agreement	1	2	288,559	1.26	0
domain_agreement	2	2	54,619	0.26	1
domain_agreement	1	1	7,170	0.03	1

Table 6.1: Link redundancy model checking runs.

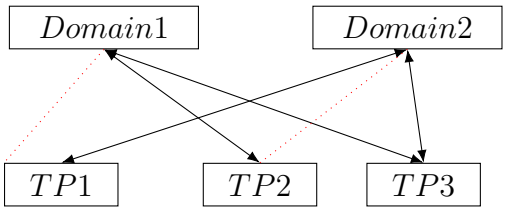

Figure 6.3: Two link failures causing the TPs to disagree on which domains are active.

6.3 Discussion

The results of verifying the link redundancy model indicate that a single link failure does not cause service packets to be dropped due to their destination being unreachable. However, two link failures can cause the TPs on the FlowSync network to disagree on which domains are active, causing packets to be dropped.

This result seems to highlight a flaw in the algorithm used to determine each domain's status. By only considering the number of peers available on each network, disregarding their identities, a split view of the network can emerge, where two domains are active according to a TP, even though not all peers are available on both domains. An immediate solution to this problem could be to store lists of available peers on each domain and compare these to decide if all peers are visible. However, this does not solve the problem of deciding which domain(s) should be active when the number of peers is equal.

Another alternative solution is to drop the concept of active and inactive domains altogether. By keeping a list of available peers on each domain, any FlowSync message can be routed to its destination as long as it is available on at least one domain. This avoids the problem of sending packets which will inevitably be lost but can lead to uneven load on the domains when a peer is only available on a single domain. In the worst case, this uneven load could cause the working domain to also fail due to insufficient bandwidth.

The problem of insufficient bandwidth on a domain seems to be illustrated by the result of setting the `THREAD_0_PRIO` constant to 1. When the Thread0 and Service processes are assigned the same priority level, verification shows a scenario where the Service process floods one of the channels, causing a deadlock. In a real implementation, this would result in packet drops, not a deadlock. However, it illustrates a scenario where an abundance of Service messages causes heartbeat packets not to be received. This situation could potentially undermine the network's operational integrity, since domains where heartbeat messages are not received, are eventually marked as inactive. In order to avoid these network overcapacity issues, care must be taken when dimensioning the bandwidth of the FlowSync network in relation to the expected volume of asymmetric internet traffic to be processed.

7

FlowSync services

This chapter covers modeling, specification, and verification of the extended model of FlowSync services, including classification, statistics synchronization, and filtering of traffic. The model is presented in Section 7.1 and verified in Section 7.2. The results are discussed in Section 7.3.

7.1 Modeling services

The simplified model of classifier sharing presented in Section 5.1 has a few limitations. Firstly, the prioritization of FlowSync messages over traffic is a stricter requirement than the RTT requirement made on the real system. Secondly, the classifier state is not modeled, and no decisions are made based on the classification of traffic. Thirdly, the statistics synchronization behavior is not modeled at all. This section presents an extended model that includes these behaviors.

```
init {
    run TP(FlowSync[0], FlowSyncTxBuf[0], Traffic[0], TrafficTxBuf
    [0]);
    run TP(FlowSync[1], FlowSyncTxBuf[1], Traffic[3], TrafficTxBuf
    [1]);

    run Tx(FlowSyncTxBuf[0], FlowSync[1], TrafficTxBuf[0], Traffic
    [1]);
    run Tx(FlowSyncTxBuf[1], FlowSync[0], TrafficTxBuf[1], Traffic
    [2]);

    run Server(Traffic[2], Traffic[0]);
    run Client(Traffic[1], Traffic[3], 0);

    run Database();
}
```

Figure 7.1: Initial process of the complete model of classifier sharing and statistics synchronization, defining the system by its component processes.

The model is extended with a `Database` process that simulates an external database. In order to relax the RTT requirement to match the real world more accurately, a special transmission buffer process, `Tx`, is introduced. The complete system is composed of two TPs with separate transmission buffer processes, an TCP/HTTP server process, a client process, and a database process. Figure 7.1 shows the complete system initialization. The full definition of the system and its component processes can be found in Appendix D.

```
proctype Tx(chan FlowIn, FlowOut, In, Out) {
    int a; mtype b; int c; int d; Stats e;
    do
        :: len(FlowIn) == 0 && len(FlowOut) == 0 && len(In) > 0;
            In ? a, b;
            Out ! a, b;
        :: FlowIn ? a, b, c, d, e;
            FlowOut ! a, b, c, d, e;
    od
}
```

Figure 7.2: Send buffer implementation

The RTT requirement can be modeled more accurately by reasoning about the order of events in the system. This has the advantage of avoiding marking each state with a timestamp, which would make otherwise identical states unique, increasing the size of the state space. What is meant by the informal requirement "the RTT of the FlowSync network is shorter than that of the traffic network" is that a FlowSync update corresponding to a traffic packet is handled before any response to that traffic packet. This can be modeled by adding a send buffer to each TP, which accepts traffic and FlowSync packets, and only forwards the traffic once all outgoing FlowSync packets have been handled. Figure 7.2 shows an implementation of such a buffer in Promela. The buffer process takes four parameters; input and output FlowSync channels, and input and output traffic channels. Traffic forwarding is only enabled when both FlowSync channels are empty, which is checked by examining the channels' lengths. This ensures that any FlowSync messages that enter the buffer before, or at the same time as, some traffic packets are handled before the traffic is forwarded. Essentially, the use of two such transmission buffers is equivalent to weakening the FlowSync priority requirement from Figure 5.5 to a directional one, only limiting traffic flow in the direction corresponding to the FlowSync messages.

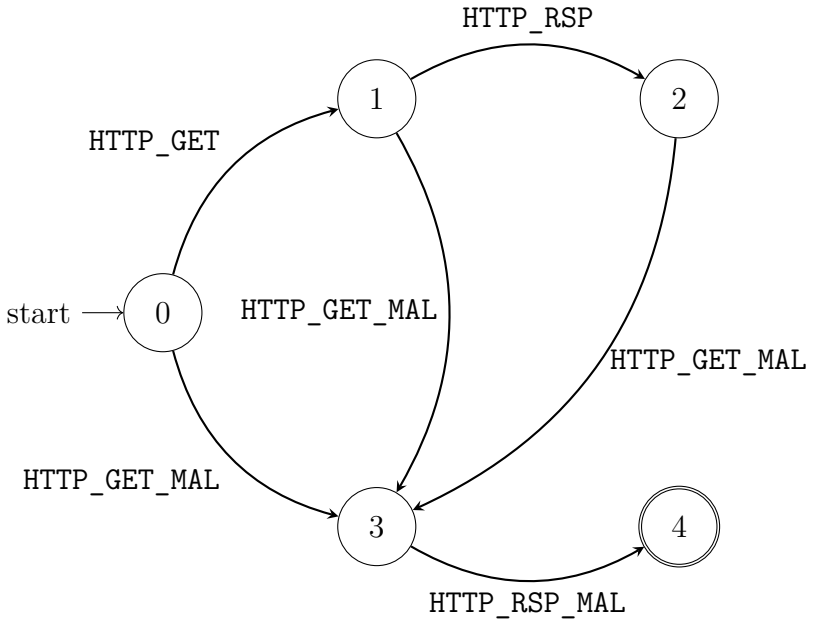

Figure 7.3: Classifier automata

The classifier state is modeled, as previously mentioned, as a simple integer. As an example, consider a classifier for HTTP traffic, similar to the one described in Section 1.2. The classifier has five states, labeled from 0 to 4. See Figure 7.3 for a state diagram of the classifier. Note that state 4 can be reached by observing a `HTTP_GET_MAL` packet followed by a `HTTP_RSP_MAL` packet. These are used to represent a malicious HTTP request, followed by a HTTP response carrying a malicious payload. The traffic packets of any HTTP conversation which enters state 4 are dropped, modeling a simple filter for malicious traffic. Note that a more trivial example could be constructed, where only observing a single malicious packet causes the connection's packets to be dropped. In this case, FlowSync is not necessary, since the traffic can be identified as malicious by only observing a single direction of flow.

```
ltl forever_fin {
    ([] ! Client[4]:sent_mal) -> ([] <> Client[4]@done)
}
ltl never_desync {
    ([] ! (TP[1]:out_of_sync || TP[2]:out_of_sync))
}
ltl never_receive_mal {
    ([] ! Client[4]:received_mal)
}

```

Figure 7.4: Safety and liveness properties of classifier sharing expressed as SPIN LTL formulas.

Finally, the model is made non-terminating by allowing the client to restart its TCP conversation with the server once it has ended. This is supported in the TP process by resetting the connection metadata structure upon seeing either an SYN or SYNACK packet.

With these extensions to the classifier sharing service, additional properties can be defined. Figure 7.4 shows the updated `forever_fin` and `never_desync` properties, as well as the new `never_receive_mal` property, which states that no client should ever receive a malicious HTTP response.

```
ltl max_down_stats {
    [] (Client[4]@done -> db_full_stats.down + db_half_stats.down
   <= real_stats.down)
}
ltl max_up_stats {
    [] (Client[4]@done -> db_full_stats.up + db_half_stats.up <=
   real_stats.up)
}

```

Figure 7.5: Safety properties of statistics synchronization expressed as SPIN LTL formulas.

As previously stated, the Promela model of classifier sharing is further extended to include the statistics synchronization behavior. The main addition is the extra FlowSync states and messages, updated to reflect those defined in Section 1.3.3, and the introduction of a `Database` process. The `Database` process listens on the new `StatsChan` channel, on which the TPs send statistics updates. The half-updates sent when a TP enters fallback mode are stored separately from the full updates sent by the upstream TP in normal mode. This is a simplification of the real database implementation, which would store the data points in a table.

The main property to be verified for statistics synchronization is the absence of statistics duplication, that is (simplified to only include packet count in each direction): "Even in the event of loss of messages on the FlowSync network, the number of packets in each traffic direction (upstream and downstream) of a connection, recorded in the external database, must not exceed the real number of packets once the connection has ended".

This property is easily expressible as an LTL formula since the actual number of packets sent in each direction can be recorded by the server and client processes, respectively, and compared to the number recorded by the database process. Equation 7.1 shows the resulting high-level LTL formula and Figure 7.5 shows the corresponding SPIN formula.

$$\Box\,(client.finished \implies (\forall\ direction \in \{upstream, downstream\}\ : database.packets[direction] \leq real.packets[direction])) \quad (7.1)$$

Since the statistics synchronization protocol is designed to be fault-tolerant, the model is also relaxed to allow the packet types `STATS_UPDATE`, `UPDATE_ACK`, `FINAL_ACK` and `FINAL_UPDATE` to be dropped by the transmission buffer process.

Property	N_REQUEST	ENABLE_CSTATE_2	States	Time (s)	Errors
forever_fin	1	true	66,595,000	280	0
never_desync	1	true	6,071,050	17.7	0
never_receive_mal	1	true	6,071,050	18.3	0
max_up_stats	1	true	6,071,050	17.8	0
max_down_stats	1	true	219,053	0.65	1
forever_fin	2	true	76,469,100	368	0
never_desync	2	true	76,783	0.33	1
never_receive_mal	2	true	76,787	0.31	1
max_up_stats	2	true	10,448,969	46.3	0
max_down_stats	2	true	259,349	1.03	1
forever_fin	2	false	69,211,200	382	0
never_desync	2	false	8,105,933	31.1	0
never_receive_mal	2	false	8,105,933	30.5	0

Table 7.1: Extended classifier sharing and statistics synchronization model checking runs.

7.2 Verification

This section presents the results of verifying the model of FlowSync services in the SPIN model checker.

Table 7.1 shows the elapsed time to verify each property with different numbers of streamed request packets. Allowing the client to send multiple requests over the same TCP connection without waiting for a response can cause out-of-sync behavior, as indicated by the failing `never_desync` and `never_receive_mal` properties. Note especially that the TPs fail to block the malicious response from the server, since subsequent classifier updates are ignored when an out-of-sync state is detected.

A message flow chart of the run causing the `never_desync` property to be violated is shown in Figure 7.6. Note that the upstream TP sees the second `HTTP_GET_MAL` request just before the downstream sees the `HTTP_RSP` response to the first request, which causes the last two UPDATE messages to overlap, illustrated by their lines intersecting in the flow chart diagram.

The results show that the modeled system cannot ensure that duplication of statistics data is impossible, shown by the `max_down_stats` property failing to be verified. Note that only duplication of downstream statistics is possible, not upstream.

See Figure 7.7 for a message flow chart showing how duplication of statistics data can occur. The duplication occurs after the final `STATS_HALF` update is sent to the database. The message is sent because both `FINAL_UPDATE` messages are dropped, and thus never acknowledged. Since the downstream TP is unaware that the upstream has already accounted and reported one of its packets to the database, it reports both of its 2 packets to the database in a half update. The sum of packets reported to the database in now 3, while the real packet count is 2.

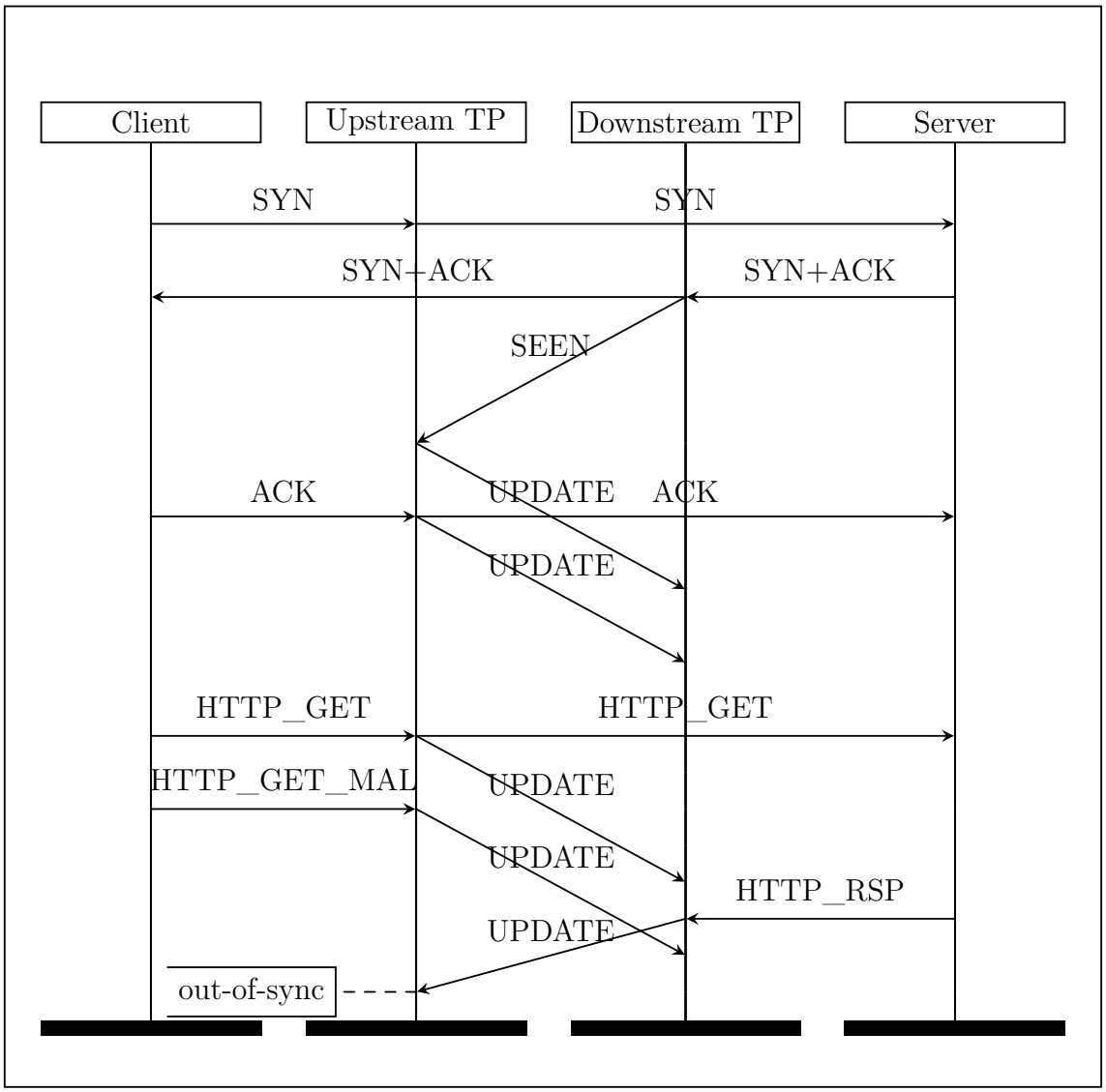

Figure 7.6: Message flow chart showing TPs going out-of-sync when a collision occurs between a FlowSync update and traffic receipt.

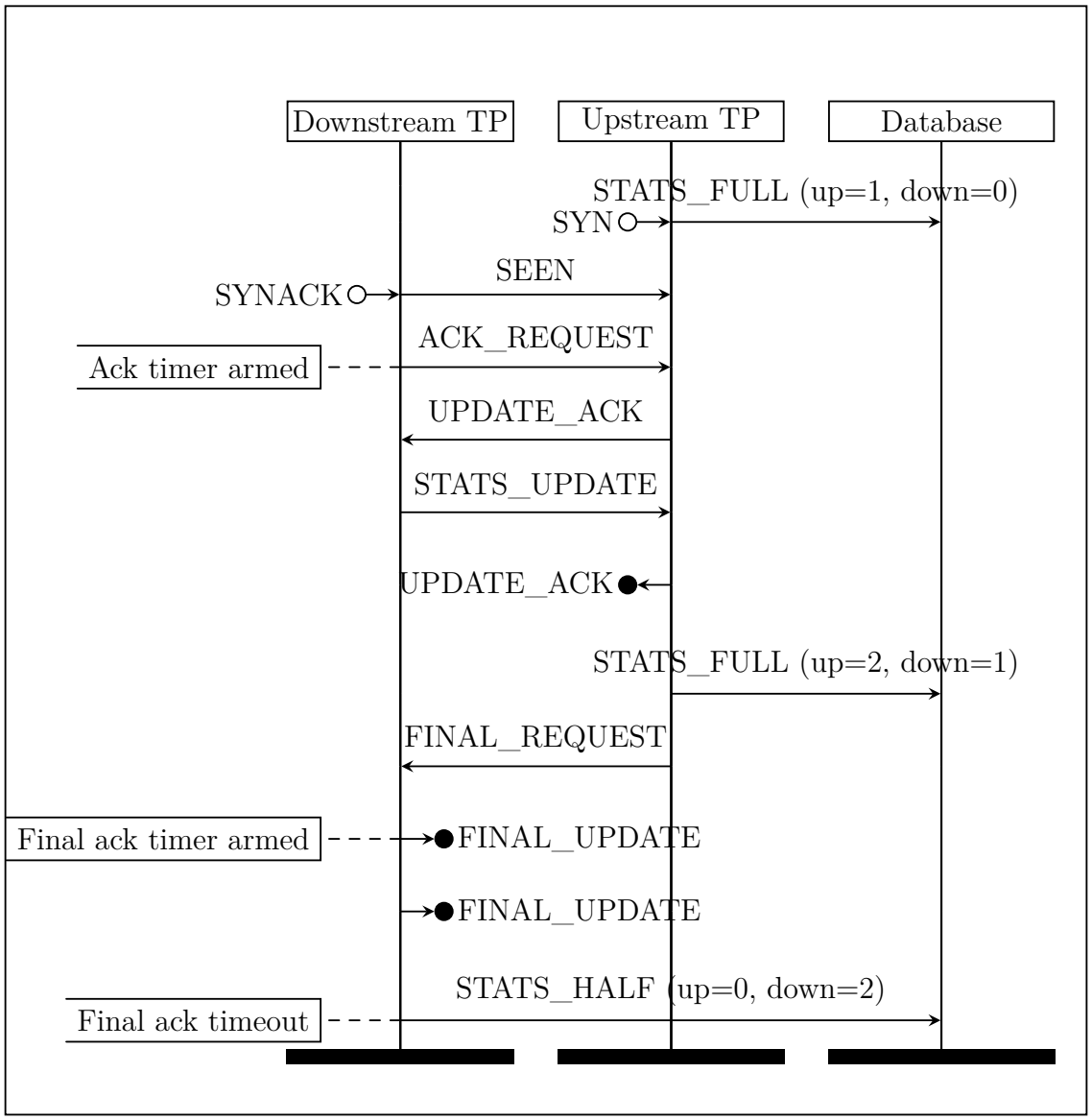

Figure 7.7: Message flow chart showing duplication of statistics data at the database due to dropped acknowledgements.

7.3 Discussion

The results show that the model of FlowSync services satisfies both the liveness and safety properties specified for classifier sharing , as long as the client waits for a response before sending additional packets. The message flow chart in Section 7.2 shows that when requests are streamed without waiting for a response, the FlowSync update triggered by the second request at the upstream TP can collide with the update triggered by the first response at the downstream TP.

The classifier state machine expects to be updated one step at a time as new data is seen inside a flow. This implicitly assumes a total order relation on the data packets which affect the state of the machine. The ordering of events in distributed systems has previously been studied, showing that a partial order on the events can be defined [39]. Note that the ordering is not total, and some pairs of events are said to be *concurrent*, meaning that it cannot be determined which event occurred "first". For some applications, it is sufficient to define a total order on the events by breaking ties by defining a total ordering of the processes involved. Since only two TPs are involved in syncing each flow, this could be implemented by, for example, declaring that a tie means that the upstream event occurred first.

In order to avoid unwanted out-of-sync states, care must be taken when deciding when to send a sequence-numbered FlowSync update. Clearly, sending an update for every traffic packet seen has the potential to cause out-of-sync in situations like the one previously shown. One option is to restrict the transitions from each state so that two different transitions from the same state cannot be triggered by packets that may be seen at either the upstream or downstream TP. An example of this is shown in Table 7.1, where out-of-sync can be avoided by removing state 2 from the classifier by setting `ENABLE_CSTATE_2` to `false`. Since the classifier does not change upon observing the HTTP response packet, no collision can occur. This result shows how the formal model can be updated to check whether a suggested solution solves the problem.

The result of the modeling and subsequent verification shows that duplicate statistics reporting is indeed possible, even with the acknowledgment-based architecture presented in Section 1.3.3. Inspecting the execution trail generated by SPIN reveals that message loss between the upstream and downstream TP can result in the downstream TP reporting statistics to the database that have already been reported by the upstream.

The proposed architecture for statistics reporting not working seems to be a consequence of the unsolvability of the Two Generals' Problem, as explained in [40]. When entering the fallback state, only one of the TPs involved (the upstream TP) knows the complete state of the system, i.e., which statistics data have been reported to the database.

This result shows two of the strengths of model checking: (1) A system which is known to be unreliable can be modeled without knowing the root cause, and the model checker can find the execution path that causes the problem. (2) Knowledge of specific computer science problems, in this case, the Two Generals' Problem, is not a prerequisite for applying model checking. The checker is not aware of the specific results of computer science research, rather, it applies the rules of formal logic mechanically to find the specific problem with our model.

A new proposed architecture for aggregating statistics at the data-plane level has been proposed at Sandvine, using TCP to facilitate communication between TPs. However, the result of the Two Generals' Problem indicates that neither TCP nor no other finite protocol can alleviate the fundamental issue of reaching a consensus over an unreliable communication channel. In spite of this, TCP is widely used to provide reliable transport over unreliable network links, and it is the base on which most of the modern internet is built.

8

Concluding remarks

This chapter concludes the thesis with a discussion of its limitations, ethical concerns, and suggestions for future work. Section 8.1 discusses the limitations of the general approach taken to verifying FlowSync. Section 8.2 touches on the ethical concerns related to the project. Conclusions are presented in Section 8.3 and suggestions future work are made in Section 8.4.

8.1 Limitations

As with any mathematical proof, the validity of the results presented in this report relies upon the validity of the assumptions. Since the traffic management system that FlowSync is a part of is large, over 100,000 lines of C code, only a fraction of the complete system is modeled in this thesis project. Therefore, the results rely on many assumptions, both implicit and explicit. A number of them are outlined in Chapter 4. This section presents and discusses limitations discovered during the implementation of the models.

For example, it is assumed that the locking of the connection table is correctly implemented, which allows us to extrapolate results regarding a single connection to many. Modeling this mechanism and verifying a model involving more connections can increase confidence in the system's correctness, at the expense of additional computational complexity. Similarly, interactions between more than 2 TPs are only considered in the model of link redundancy, based on the assumption that the network is configured in such a way that no more than 2 TPs in the same FlowSync network will ever see the same connection.

While modeling the service and link redundancy layers separately is beneficial from a state space point-of-view, it also introduces a few limitations: (1) The specific interaction between the messages sent by the services and the heartbeat packets is not modeled; instead, the representative service process in the link redundancy model can send messages at any time. However, it could be argued that this more general model provides a stronger guarantee for the link layer's correctness than would otherwise be possible. (2) Likewise, potential fairness issues between the link redundancy heartbeat process (Thread0) and the service processes cannot be reasoned about. However, such issues are unlikely to exist in the real system, where the service processes and heartbeat process are pinned to separate CPU cores. As previously shown, the link redundancy model allows reasoning about fairness issues related to the network bandwidth, modeled using buffered channels of a fixed size.

8.2 Ethical concerns

Since this thesis work deals with modeling and verifying the correct operation of software for analyzing and managing network traffic, the possible use cases for the results of the project are many. Internet traffic management can be used for planning network infrastructure and capacity, balancing bandwidth usage, and enforcing legal policy, either through blocking illegal content or wiretapping suspected criminals. However, in the wrong hands, the same features that allow these uses can be used for censorship and invasion of privacy.

8.3 Conclusion

In this thesis, we have shown how three parts (link redundancy, classifier sharing, and statistics synchronization) of a system for the management of asymmetric internet traffic can be modeled in a formal language, and how this model can be verified against safety and liveness properties to identify issues in the design. Furthermore, we have discussed the relationship between these issues and corresponding known problems in distributed systems, e.g. the Two Generals' Problem and the problem of establishing a total order of distributed events. Lastly, we have provided suggestions for how to work around the limitations imposed by these problems, to avoid issues in the real-world implementation of the design.

We approached the problem of modeling and verifying a real system without a specific modeling framework in mind. By attempting modeling in three separate frameworks, we identified what features would be required to accurately model FlowSync. We hope that his approach can also be used in future work on modeling real distributed systems, where the requirements may not be known beforehand.

Previous research has demonstrated the usefulness of formal methods in verifying Software Defined Networking (SDN) systems and distributed data structure protocols. In this thesis project, we have further demonstrated their applicability to distributed data-plane traffic management systems. The framework used is shown to permit reasoning about the network both in terms of the internal state of the traffic processing nodes and from a network client's point-of-view, where we verify both liveness and safety properties.

8.4 Future work

In the discussion of the classifier sharing results, a potential link to previous research on establishing a total order of events in a distributed system was discovered. The findings presented there could serve as a basis for designing a distributed traffic classification protocol that does not suffer from concurrent classifier updates causing out-of-sync behavior. A future project could consist of implementing a version of FlowSync that incorporates the methods presented in a formal language and verifying it against the same properties that failed to be verified in this project.

In order to gain further confidence in the correct operation of the FlowSync protocol implementation, additional verification techniques can be used. These include (1) runtime verification [41], which involves analysis of the execution traces of the real implementation rather than an exhaustive state-space search, and (2) running the implementation in a modeled environment using the C Model Checker (CMC) [27].

The survey of modeling frameworks conducted as part of this project identified a few disparities in the set of features provided by each framework. We think that this can serve as directions for future work on implementing features from other frameworks in TLA$^+$, R-CHECK, and SPIN. For example, the backend of R-CHECK, nuXmv, supports array types [42], which could allow for the implementation of array types into R-CHECK. Another example is to introduce quantifiers such as *forall* ($\forall$) and *exists* ($\exists$) into Promela, to avoid unnecessary use of macros or code repetition.

Bibliography

[1] J. Postel, "User datagram protocol," RFC Editor, STD 6, Aug. 1980, http://www.rfc-editor.org/rfc/rfc768.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc768.txt.

[2] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, 2022. DOI: 10.17487/RFC9293. [Online]. Available: https://www.rfc-editor.org/info/rfc9293.

[3] Z. Hou, Y. Huang, S. Zheng, X. Dong, and B. Wang, "Design and implementation of heartbeat in multi-machine environment," in *17th International Conference on Advanced Information Networking and Applications, 2003. AINA 2003.*, IEEE, 2003, pp. 583–586.

[4] P. Brodsky, *Internet traffic and capacity remain brisk*, https://blog.telegeography.com/internet-traffic-and-capacity-remain-brisk, Accessed: 2022-12-06, Sep. 13, 2022.

[5] *De-cix frankfurt traffic statistics*, https://www.de-cix.net/en/locations/frankfurt/statistics, Accessed: 2022-12-10, Dec. 10, 2022.

[6] *Ams-ix - total traffic*, https://stats.ams-ix.net/index.html, Accessed: 2022-12-10, Dec. 10, 2022.

[7] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker, "On routing asymmetry in the internet," in *GLOBECOM'05. IEEE Global Telecommunications Conference, 2005.*, IEEE, vol. 2, 2005, 6–pp.

[8] R. Braden, "Requirements for internet hosts - communication layers," RFC Editor, STD 3, Oct. 1989, http://www.rfc-editor.org/rfc/rfc1122.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1122.txt.

[9] G. Bochmann and C. Sunshine, "Formal methods in communication protocol design," *IEEE transactions on Communications*, vol. 28, no. 4, pp. 624–631, 1980.

[10] L. Schumann, T. V. Doan, T. Shreedhar, R. Mok, and V. Bajpai, "Impact of evolving protocols and covid-19 on internet traffic shares," *arXiv preprint arXiv:2201.00142*, 2022.

[11] S. Zander, T. Nguyen, and G. Armitage, "Self-learning ip traffic classification based on statistical flow characteristics," in *Passive and Active Network Measurement: 6th International Workshop, PAM 2005, Boston, MA, USA, March 31-April 1, 2005. Proceedings 6*, Springer, 2005, pp. 325–328.

[12] *Iq52600 platform*, Accessed: 2023-04-06, Sandvine Corporation, 2022. [Online]. Available: https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2021/Datasheets/Sandvine_DS_iQ52600%20-%20CONFIDENTIAL.pdf.

[13] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[14] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé, "Using symbolic execution for verifying safety-critical systems," *SIGSOFT Softw. Eng. Notes*, pp. 142–151, 2001. DOI: 10.1145/503271.503230. [Online]. Available: https://doi.org/10.1145/503271.503230.

[15] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Computer Science Review*, vol. 5, no. 2, pp. 163–203, 2011.

[16] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, pp. 1–54, 2009.

[17] P. Schnoebelen, "The complexity of temporal logic model checking.," *Advances in modal logic*, vol. 4, no. 393-436, p. 35, 2002.

[18] C. S. T. Wien, C. Schneidewind, T. Wien, *et al.*, *Ethor: Practical and provably sound static analysis of ethereum smart contracts: Proceedings of the 2020 acm sigsac conference on computer and communications security*, 2020. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3372297.3417250.

[19] C. Schneidewind, M. Scherer, and M. Maffei, *The good, the bad and the ugly: Pitfalls and best practices in automated sound static analysis of ethereum smart contracts*, 2020. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-61467-6_14.

[20] F. Chen and G. Roşu, "Mop: An efficient and generic runtime verification framework," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, 2007, pp. 569–588.

[21] C. Colombo, G. J. Pace, and G. Schneider, "Larva — safer monitoring of real-time java programs (tool paper)," in *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, 2009, pp. 33–37. DOI: 10.1109/SEFM.2009.13.

[22] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, ieee, 1977, pp. 46–57.

[23] L. Lamport, "Specifying concurrent program modules," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 5, no. 2, pp. 190–222, 1983.

[24] S. Berezin, S. Campos, and E. M. Clarke, "Compositional reasoning in model checking," *Lecture Notes in Computer Science*, vol. 1536, pp. 81–102, 1998.

[25] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 483–494, 2013.

[26] M. Musuvathi, D. R. Engler, *et al.*, "Model checking large network protocol implementations.," in *NSDI*, vol. 4, 2004, pp. 12–12.

[27] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, "{Cmc}: A pragmatic approach to model checking real code," in *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, 2002.

[28] R. Gerth, *Concise promela reference*, 1997. [Online]. Available: `https://spinroot.com/spin/Man/Quick.html`.

[29] T. C. Ruys, *Spin tutorial - spinroot.com*, 2002. [Online]. Available: `https://spinroot.com/spin/Doc/SpinTutorial.pdf`.

[30] G. Costa and C. Stirling, "Weak and strong fairness in ccs," *Information and Computation*, vol. 73, no. 3, pp. 207–244, 1987.

[31] D. Bošnački, "A light-weight algorithm for model checking with symmetry reduction and weak fairness," in *Model Checking Software: 10th International SPIN Workshop Portland, OR, USA, May 9–10, 2003 Proceedings 10*, Springer, 2003, pp. 89–103.

[32] G. J. Holzmann, *The spin model checker: Manual*, Bell Laboratories, 2017. [Online]. Available: `https://spinroot.com/spin/Man/Manual.html`.

[33] L. Lamport, "Specifying systems: The tla+ language and tools for hardware and software engineers," 2002.

[34] T. Lu, S. Merz, and C. Weidenbach, "Towards verification of the pastry protocol using tla+," in *Formal Techniques for Distributed Systems*, Springer, 2011, pp. 244–258.

[35] J. A. Lund, "Verification of the chord protocol in tla+," M.S. thesis, UiT Norges arktiske universitet, 2019.

[36] Y. A. Alrahman, S. Azzopardi, and N. Piterman, "R-CHECK: A model checker for verifying reconfigurable MAS," in *21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022, Auckland, New Zealand, May 9-13, 2022*, P. Faliszewski, V. Mascardi, C. Pelachaud, and M. E. Taylor, Eds., International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), 2022, pp. 1518–1520. DOI: `10.5555/3535850.3536020`. [Online]. Available: `https://www.ifaamas.org/Proceedings/aamas2022/pdfs/p1518.pdf`.

[37] Y. Abd Alrahman, S. Azzopardi, and N. Piterman, "Model checking reconfigurable interacting systems," in *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning: 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22–30, 2022, Proceedings, Part III*, Rhodes, Greece: Springer-Verlag, 2022, pp. 373–389, ISBN: 978-3-031-19758-1. DOI: `10.1007/978-3-031-19759-8_23`. [Online]. Available: `https://doi.org/10.1007/978-3-031-19759-8_23`.

[38] R. Cavada, A. Cimatti, M. Dorigatti, *et al.*, "The nuxmv symbolic model checker," in *CAV*, 2014, pp. 334–342.

[39] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications*, 1978.

[40] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber, "Some constraints and tradeoffs in the design of network communications," in *Proceedings of the fifth ACM symposium on Operating systems principles*, 1975, pp. 67–74.

[41] E. Bartocci and Y. Falcone, *Lectures on Runtime Verification*. Springer, 2018.

[42] M. Bozzano, R. Cavada, A. Cimatti, *et al.*, "Nuxmv 2.0. 0 user manual," *Fondazione Bruno Kessler, Tech. Rept., Trento, Italy*, 2019.

A

Classifier sharing model in Promela

```
mtype = {
    /* TCP */
    SYN, SYNACK, ACK, DATA, FIN,
    /* FlowSync */
    SEEN, UPDATE, ENDED,
    /* States */
    FS_INIT, FS_SEEN, FS_SYNCING, FS_ENDED
};

chan FlowSync[2] = [100] of {mtype, int};
chan Traffic[4] = [100] of {mtype, int};

proctype TP(chan FlowIn, FlowOut, Upstream, Downstream) {
    // FlowSync channels are accessed by the never claim
   corresponding to the
    // LTL formulas, so partial order reduction is not possible.
    //xr FlowIn;
    //xs FlowOut;

    xr Upstream;
    xs Downstream;

    mtype state;
    bool out_of_sync;
    bool not_seen;

    int flow_seq = 0;

    mtype type; int seq;

    do
        // Handle Traffic packet
        :: Upstream ? type, seq;
handle_traffic:
            d_step {
            if
                :: type == SYN;
                    state = FS_SEEN;

                :: type == SYNACK;
                    printf("Syncing\n");
                    state = FS_SYNCING;
                    FlowOut ! SEEN, 0;
```

```
43
44                  :: type == DATA || type == ACK;
45                      FlowOut ! UPDATE, flow_seq;
46                      flow_seq++;
47
48                  :: type == FIN;
49                      state = FS_ENDED;
50                      printf("FIN %d\n", flow_seq);
51                      FlowOut ! ENDED, flow_seq;
52                      flow_seq++;
53
54                  :: else; skip;
55              fi
56
57              Downstream ! type, seq;
58              }
59
60          // Handle FlowSync packet.
61          :: FlowIn ? type, seq;
62              d_step {
63
64              printf("Saw flowsync: %d\n", seq);
65              if
66                  :: seq != flow_seq && state != FS_ENDED;
67                      printf("Out of sync: seq = %d, flow_seq = %d\n"
    , seq, flow_seq);
68                      out_of_sync = true;
69                  :: else; skip;
70              fi
71
72              if
73                  :: state == FS_SYNCING && type == UPDATE;
74                      flow_seq++;
75                  :: state == FS_SEEN && type == SEEN;
76                      printf("Syncing\n");
77                      state = FS_SYNCING;
78                      FlowOut ! UPDATE, flow_seq;
79                      flow_seq++;
80                  :: type == ENDED;
81                      state = FS_ENDED;
82                      flow_seq++;
83                  :: else;
84                      not_seen = true;
85              fi
86              }
87      od
88  }
89
90  proctype Server(chan In, Out) {
91      xr In;
92      xs Out;
93      mtype type; int seq;
94
95      // TCP handshake.
96      In ? SYN, seq;
97      Out ! SYNACK, 1;
```

```
98
99      // Well-behaved TCP waits for ACK from client before sending
        DATA.
100     // Removing this line allows out-of-sync when the server sends
        DATA
101     // directly following SYNACK, since the FlowSync SEEN-UPDATE
        handshake
102     // has not occurred yet.
103     In ? ACK, seq;
104
105     // Pick a seq, any seq.
106     int server_seq = 10;
107
108     do
109         // Receive request packets.
110         :: In ? type, seq;
111             printf("Server Received %d\n", seq);
112             if
113                 :: type == FIN; break;
114                 :: else; skip;
115             fi
116         // Randomly send response packets.
117         :: server_seq < 14;
118             printf("Server sending %d\n", server_seq);
119             Out ! DATA, server_seq;
120             server_seq++;
121     od
122
123     // Termination handshake.
124     Out ! ACK, server_seq;
125     Out ! FIN, server_seq + 1;
126     In ? ACK, seq;
127
128     printf("Server done.\n");
129 done:
130 }
131
132 proctype Client(chan In, Out) {
133     xr In;
134     xs Out;
135     mtype type; int seq;
136
137     // TCP handshake.
138     Out ! SYN, 0;
139     In ? SYNACK, seq;
140     Out ! ACK, seq + 1;
141
142     int count = 4;
143
144     do
145         ::
146         Out ! DATA, seq + 1;
147         count = count - 1;
148         if
149             :: count == 0; break;
150             :: else -> skip;
```

```
151         fi
152         :: In ? type, seq;
153         printf("Client Received %d\n", seq);
154     od
155
156     // Termination handshake.
157     Out ! FIN, seq + 1;
158
159     // Wait for ACK.
160     do
161         :: In ? type, seq;
162             if
163                 :: type == ACK;
164                     break;
165                 :: else; skip;
166             fi
167     od
168
169     In ? FIN, seq;
170     Out ! ACK, seq + 1;
171
172     printf("Client done.\n");
173 done:
174 }
175
176 init {
177     run TP(FlowSync[0], FlowSync[1], Traffic[0], Traffic[1]);
178     run TP(FlowSync[1], FlowSync[0], Traffic[3], Traffic[2]);
179
180     run Server(Traffic[2], Traffic[0]);
181     run Client(Traffic[1], Traffic[3]);
182 }
183
184 // Ensure that FlowSync messages are handled before traffic.  This
    is necessary
185 // to guarantee absence of out-of-sync.
186 #define FLOWSYNC_PRIO(i) (TP[i]@handle_traffic -> (len(FlowSync[i -
    1]) == 0))
187 #define FLOWSYNC_PRIO_ALL ([] (FLOWSYNC_PRIO(1) && FLOWSYNC_PRIO(2)
    ))
188
189 ltl eventually_sync { FLOWSYNC_PRIO_ALL -> (<> (TP[1]:state ==
    FS_SYNCING && TP[2]:state == FS_SYNCING)) }
190 ltl eventually_fin { FLOWSYNC_PRIO_ALL -> (<> Client@done && <>
    Server@done) }
191 ltl never_desync { FLOWSYNC_PRIO_ALL -> ([] ! (TP[1]:out_of_sync ||
    TP[2]:out_of_sync)) }
```

B

Classifier sharing model in R-CHECK

```
channels: flowsync1, flowsync2, external1, external2, i1, i2

enum msgvals {SYN, SACK, ACK, DATA, FIN, SEEN, UPDATE, ENDED}

enum tcpstate {listening, connecting, connected, closing, closed}

message-structure: MSG : msgvals, SEQ : int
communication-variables: nop : int

agent Client
    local: tstate : tcpstate, seq : int, in : channel, out : channel
    init: tstate == connecting && seq == 20
    relabel:
        nop <- 0
    receive-guard: (channel == in)

    repeat: (
                (
                    <tstate == connecting> out ! (TRUE) (MSG := SYN, SEQ := seq) [seq := seq + 1];
                    <tstate == connecting && MSG == SACK> in ? [];
                    <TRUE> out ! (TRUE) (MSG := ACK, SEQ := seq) [tstate := connected, seq := seq + 1]
                )
                +
                <tstate == connected && seq < 24> out ! (TRUE) (MSG := DATA, SEQ := seq) [seq := seq + 1]
                +
                <tstate == connected && MSG == DATA> in ? []
                +
                (
                    <tstate == connected && seq == 24> out ! (TRUE) (MSG := FIN, SEQ := seq) [seq := seq + 1];
                    <MSG == ACK> in ? []
                )
                +
                (
                    <tstate == closing && MSG == FIN> in ? [tstate := closed];
```

```
35                      <TRUE> out ! (TRUE) (MSG := ACK, SEQ := seq) [
    seq := seq + 1]
36                  )
37          )
38
39 agent Server
40      local: tstate : tcpstate, seq : int, in : channel, out :
    channel
41      init: tstate == listening && seq == 10
42      relabel:
43          nop <- 0
44      receive-guard: (channel == in)
45
46      repeat: (
47                  (
48                      <tstate == listening & MSG == SYN> in ? [tstate
     := connecting];
49                      <TRUE> out ! (TRUE) (MSG := SACK, SEQ := seq) [
    seq := seq + 1]
50                  )
51                  +
52                  <tstate == connecting && MSG == ACK> in ? [tstate
    := connected]
53                  +
54                  <tstate == connected && seq < 14> out ! (TRUE) (MSG
     := DATA, SEQ := seq) [seq := seq + 1]
55                  +
56                  <tstate == connected && MSG == DATA> in ? []
57                  +
58                  (
59                      <tstate == connected & MSG == FIN> in ? [tstate
     := closing];
60                      <TRUE> out ! (TRUE) (MSG := ACK, SEQ := seq) [
    seq := seq + 1];
61                      <TRUE> out ! (TRUE) (MSG := FIN, SEQ := seq) [
    seq := seq + 1];
62                      <TRUE & MSG == ACK> in ? [tstate := closed]
63                  )
64              )
65
66 agent Engine
67      local: flowin : channel, flowout : channel, in : channel, out :
     channel, seen : bool, syncing : bool, ended : bool, outofsync :
     bool, flowseq : int, fwdtype : msgvals, fwdseq : int
68      init: seen == FALSE && syncing == FALSE && ended == FALSE &&
    outofsync == FALSE && fwdtype == SYN && flowseq == 0
69      relabel:
70          nop <- 0
71      receive-guard: (channel == in) | (channel == flowin)
72
73      repeat: (
74                  (
75                      <MSG == SYN> in ? [fwdseq := SEQ, seen := TRUE
    ];
76                      <TRUE> out ! (TRUE) (MSG := SYN, SEQ := fwdseq)
     []
```

```
77                  )
78                  +
79                  (
80                      <MSG == SACK> in ? [fwdseq := SEQ, syncing :=
    TRUE];
81                      <TRUE> flowout ! (TRUE) (MSG := SEEN, SEQ :=
    flowseq) [];
82                      <TRUE> out ! (TRUE) (MSG := SACK, SEQ := fwdseq
    ) []
83                  )
84                  +
85                  (
86                      <MSG == DATA | MSG == ACK> in ? [fwdtype := MSG
    , fwdseq := SEQ];
87                      <TRUE> flowout ! (TRUE) (MSG := UPDATE, SEQ :=
    flowseq) [flowseq := flowseq + 1];
88                      <TRUE> out ! (TRUE) (MSG := fwdtype, SEQ :=
    fwdseq) []
89                  )
90                  +
91                  (
92                      <MSG == FIN> in ? [fwdtype := MSG, fwdseq :=
    SEQ, ended := TRUE];
93                      <TRUE> flowout ! (TRUE) (MSG := ENDED, SEQ :=
    flowseq) [flowseq := flowseq + 1];
94                      <TRUE> out ! (TRUE) (MSG := fwdtype, SEQ :=
    fwdseq) []
95                  )
96                  +
97                  (
98                      <MSG == UPDATE> flowin ? [flowseq := flowseq +
    1, outofsync := (SEQ != flowseq)]
99                  )
100                 +
101                 (
102                     <MSG == SEEN & seen> flowin ? [syncing := TRUE,
     outofsync := (SEQ != flowseq)];
103                     <TRUE> flowout ! (TRUE) (MSG := UPDATE, SEQ :=
    flowseq) [flowseq := flowseq + 1]
104                 )
105                 +
106                 (
107                     <MSG == ENDED> flowin ? [ended := TRUE, flowseq
     := flowseq + 1, outofsync := (SEQ != flowseq)]
108                 )
109             )
110
111 system = Engine(engine1, flowin == flowsync1 && flowout ==
    flowsync2 && out == external1 && in == i1) | Engine(engine2,
    flowin == flowsync2 && flowout == flowsync1 && out == i2 && in
    == external2) | Server(server, in == external1 && out ==
    external2) | Client(client, in == i2 && out == i1)
112
113 SPEC F (engine1-syncing && engine2-syncing)
114 SPEC F (server-tstate == closed && client-tstate == closed)
115 SPEC G (!engine1-outofsync && !engine2-outofsync)
```

C

Classifier sharing model in TLA$^+$

```
1  ---------------------------- MODULE Classification
       ----------------------------
2
3  EXTENDS Integers, Sequences, TLC, FiniteSets
4
5  CONSTANTS MaxQueue, MaxSeq, NULL
6
7  SeqOf(set, n) == UNION {[1..m -> set] : m \in 0..n}
8  seq (+) elem == Append(seq, elem)
9
10 MessageType == {"SYN", "SYNACK", "ACK", "DATA", "FIN", "SEEN", "
       UPDATE", "ENDED"}
11 ChannelNames == {"flowsync1", "flowsync2", "ext1", "int1", "ext2",
       "int2"}
12 Clients == {"client"}
13 Servers == {"server"}
14 TPs == {"TP1", "TP2"}
15 Boxes == Clients \union Servers \union TPs
16
17 set ++ e == set \union {e}
18 Messages == [seq: 0..MaxSeq, msg: MessageType]
19 EmptyPacket == [seq |-> 0, msg |-> "DATA"]
20
21
22 (*--fair algorithm classification
23 variables
24     channels = [name \in ChannelNames |-> <<>>];
25     flowin = [tp \in TPs |-> IF tp = "TP1" THEN "flowsync1" ELSE "
       flowsync2"];
26     flowout = [tp \in TPs |-> IF tp = "TP2" THEN "flowsync1" ELSE "
       flowsync2"];
27     in = [box \in Boxes |-> IF box = "TP1" THEN "int1" ELSE
28                             IF box = "TP2" THEN "ext2" ELSE
29                             IF box = "client" THEN "int2" ELSE
30                             "ext1"];
31     out = [box \in Boxes |-> IF box = "TP1" THEN "ext1" ELSE
32                              IF box = "TP2" THEN "int2" ELSE
33                              IF box = "client" THEN "int1" ELSE
34                              "ext2"];
35
36 define
37     TypeInvariant ==
```

```
        \A channel \in channels : channel \in SUBSET SeqOf(Messages
    , MaxQueue)
end define;

macro Send(name, msg) begin
    channels[name] := @ (+) msg;
end macro;

macro Receive(name, msg) begin
    await channels[name] /= <<>>;
    msg := Head(channels[name]);
    channels[name] := Tail(channels[name]);
end macro;

process TP \in TPs
variables
    d = EmptyPacket;
    seen = FALSE;
    syncing = FALSE;
    ended = FALSE;
    outofsync = FALSE;
    flowseq = 0;
begin
TP_Loop:
    while TRUE do
        either
            Receive(in[self], d);
TP_traffic_handle:
            if d.msg = "SYN" then
                seen := TRUE;
            elsif d.msg = "SYNACK" then
                syncing := TRUE;
                Send(flowout[self], [msg |-> "SEEN", seq |-> 0]);
            elsif d.msg \in {"DATA", "ACK"} then
                Send(flowout[self], [msg |-> "UPDATE", seq |->
    flowseq]);
                flowseq := flowseq + 1;
            elsif d.msg = "FIN" then
                ended := TRUE;
                Send(flowout[self], [msg |-> "ENDED", seq |->
    flowseq]);
                flowseq := flowseq + 1;
            end if;
TP_traffic_forward:
            Send(out[self], d);
        or
            Receive(flowin[self], d);
TP_flowsync_handle:
            if (d.seq /= flowseq) /\ ~ended then
                outofsync := TRUE;
            end if;

            if syncing /\ d.msg = "UPDATE" then
                flowseq := flowseq + 1;
            elsif seen /\ d.msg = "SEEN" then
                syncing := TRUE;
```

```
                Send(flowout[self], [msg |-> "UPDATE", seq |-> flowseq]);
                flowseq := flowseq + 1;
            elsif d.msg = "ENDED" then
                ended := TRUE;
                flowseq := flowseq + 1;
            end if;
        end either;
    end while;
end process;

process client \in Clients
variables
    d = EmptyPacket;
    client_seq = 10;

begin
Client_sSYN:
    Send(out[self], [seq |-> client_seq, msg |-> "SYN"]);
    client_seq := client_seq + 1;

Client_rSYNACK:
    Receive(in[self], d);
    assert d.msg = "SYNACK";

Client_sACK:
    Send(out[self], [seq |-> client_seq, msg |-> "ACK"]);
    client_seq := client_seq + 1;

Client_DataLoop:
    while client_seq < 14 do
Client_Data:
        either
            Receive(in[self], d);
        or
            Send(out[self], [seq |-> client_seq, msg |-> "DATA"]);
            client_seq := client_seq + 1;
        end either;
    end while;

Client_sFIN:
    Send(out[self], [seq |-> client_seq, msg |-> "FIN"]);
    client_seq := client_seq + 1;

Client_rFINACK:
    while TRUE do
        Receive(in[self], d);
        if d.msg = "ACK" then
            goto Client_rFIN;
        end if;
    end while;

Client_rFIN:
    Receive(in[self], d);
    assert d.msg = "FIN";

```

```
146 Client_sFINACK:
147     Send(out[self], [seq |-> client_seq, msg |-> "ACK"]);
148     client_seq := client_seq + 1;
149 
150 end process;
151 
152 process server \in Servers
153 variables
154     d = EmptyPacket;
155     server_seq = 20;
156 begin
157 Server_rSYN:
158     Receive(in[self], d);
159     assert d.msg = "SYN";
160 
161 Server_sSYNACK:
162     Send(out[self], [seq |-> server_seq, msg |-> "SYNACK"]);
163     server_seq := server_seq + 1;
164 
165 Server_rACK:
166     Receive(in[self], d);
167     assert d.msg = "ACK";
168 
169 Server_Data_Loop:
170     while d.msg /= "FIN" do
171 Server_Data:
172         either
173             Receive(in[self], d);
174         or
175             await server_seq < 24;
176             Send(out[self], [seq |-> server_seq, msg |-> "DATA"]);
177             server_seq := server_seq + 1;
178         end either;
179     end while;
180 
181 Server_sFINACK:
182     Send(out[self], [seq |-> server_seq, msg |-> "ACK"]);
183     server_seq := server_seq + 1;
184 
185 Server_sFIN:
186     Send(out[self], [seq |-> server_seq, msg |-> "FIN"]);
187     server_seq := server_seq + 1;
188 
189 Server_rFINACK:
190     Receive(in[self], d);
191     assert d.msg = "ACK";
192 
193 end process;
194 
195 end algorithm; *)
196 \* BEGIN TRANSLATION (chksum(pcal) = "57eef57" /\ chksum(tla) = "31
       bc4f99")
197 \* Process variable d of process TP at line 53 col 5 changed to d_
198 \* Process variable d of process client at line 103 col 5 changed
       to d_c
199 VARIABLES channels, flowin, flowout, in, out, pc
```

```
200
201 (* define statement *)
202 TypeInvariant ==
203     \A channel \in channels : channel \in SUBSET SeqOf(Messages,
    MaxQueue)
204
205 VARIABLES d_, seen, syncing, ended, outofsync, flowseq, d_c,
    client_seq, d,
206           server_seq
207
208 vars == << channels, flowin, flowout, in, out, pc, d_, seen,
    syncing, ended,
209            outofsync, flowseq, d_c, client_seq, d, server_seq >>
210
211 ProcSet == (TPs) \cup (Clients) \cup (Servers)
212
213 Init == (* Global variables *)
214         /\ channels = [name \in ChannelNames |-> <<>>]
215         /\ flowin = [tp \in TPs |-> IF tp = "TP1" THEN "flowsync1"
    ELSE "flowsync2"]
216         /\ flowout = [tp \in TPs |-> IF tp = "TP2" THEN "flowsync1"
     ELSE "flowsync2"]
217         /\ in = [box \in Boxes |-> IF box = "TP1" THEN "int1" ELSE
218                                    IF box = "TP2" THEN "ext2" ELSE
219                                    IF box = "client" THEN "int2"
    ELSE
220                                    "ext1"]
221         /\ out = [box \in Boxes |-> IF box = "TP1" THEN "ext1" ELSE
222                                     IF box = "TP2" THEN "int2" ELSE
223                                     IF box = "client" THEN "int1"
    ELSE
224                                     "ext2"]
225         (* Process TP *)
226         /\ d_ = [self \in TPs |-> EmptyPacket]
227         /\ seen = [self \in TPs |-> FALSE]
228         /\ syncing = [self \in TPs |-> FALSE]
229         /\ ended = [self \in TPs |-> FALSE]
230         /\ outofsync = [self \in TPs |-> FALSE]
231         /\ flowseq = [self \in TPs |-> 0]
232         (* Process client *)
233         /\ d_c = [self \in Clients |-> EmptyPacket]
234         /\ client_seq = [self \in Clients |-> 10]
235         (* Process server *)
236         /\ d = [self \in Servers |-> EmptyPacket]
237         /\ server_seq = [self \in Servers |-> 20]
238         /\ pc = [self \in ProcSet |-> CASE self \in TPs -> "TP_Loop
    "
239                                       [] self \in Clients -> "
    Client_sSYN"
240                                       [] self \in Servers -> "
    Server_rSYN"]
241
242 TP_Loop(self) == /\ pc[self] = "TP_Loop"
243                  /\ \/ /\ channels[(in[self])] /= <<>>
244                        /\ d_' = [d_ EXCEPT ![self] = Head(channels
    [(in[self])])]
```

```
245                         /\ channels' = [channels EXCEPT ![(in[self])
    ] = Tail(channels[(in[self])])]
246                         /\ pc' = [pc EXCEPT ![self] = "
    TP_traffic_handle"]
247                      \/ /\ channels[(flowin[self])] /= <<>>
248                         /\ d_' = [d_ EXCEPT ![self] = Head(channels
    [(flowin[self])])]
249                         /\ channels' = [channels EXCEPT ![(flowin[
    self])] = Tail(channels[(flowin[self])])]
250                         /\ pc' = [pc EXCEPT ![self] = "
    TP_flowsync_handle"]
251                  /\ UNCHANGED << flowin, flowout, in, out, seen,
    syncing,
252                                  ended, outofsync, flowseq, d_c,
    client_seq, d,
253                                  server_seq >>
254
255 TP_traffic_handle(self) == /\ pc[self] = "TP_traffic_handle"
256                            /\ IF d_[self].msg = "SYN"
257                                  THEN /\ seen' = [seen EXCEPT ![
    self] = TRUE]
258                                       /\ UNCHANGED << channels,
    syncing, ended,
259                                                       flowseq >>
260                                  ELSE /\ IF d_[self].msg = "SYNACK"
261                                             THEN /\ syncing' = [
    syncing EXCEPT ![self] = TRUE]
262                                                  /\ channels' = [
    channels EXCEPT ![(flowout[self])] = @ (+) ([msg |-> "SEEN", seq
     |-> 0])]
263                                                  /\ UNCHANGED <<
    ended,
264
    flowseq >>
265                                             ELSE /\ IF d_[self].msg
     \in {"DATA", "ACK"}
266                                                     THEN /\
    channels' = [channels EXCEPT ![(flowout[self])] = @ (+) ([msg
    |-> "UPDATE", seq |-> flowseq[self]])]
267                                                          /\
    flowseq' = [flowseq EXCEPT ![self] = flowseq[self] + 1]
268                                                          /\
    ended' = ended
269                                                     ELSE /\ IF
    d_[self].msg = "FIN"
270
    THEN /\ ended' = [ended EXCEPT ![self] = TRUE]
271
        /\ channels' = [channels EXCEPT ![(flowout[self])] = @ (+)
    ([msg |-> "ENDED", seq |-> flowseq[self]])]
272
        /\ flowseq' = [flowseq EXCEPT ![self] = flowseq[self] + 1]
273
    ELSE /\ TRUE
274
        /\ UNCHANGED << channels,
```

```
275
                        ended,
276
                        flowseq >>
277                                              /\ UNCHANGED
    syncing
278                                         /\ seen' = seen
279                             /\ pc' = [pc EXCEPT ![self] = "
    TP_traffic_forward"]
280                             /\ UNCHANGED << flowin, flowout, in, out
    , d_,
281                                             outofsync, d_c,
    client_seq, d,
282                                             server_seq >>
283
284 TP_traffic_forward(self) == /\ pc[self] = "TP_traffic_forward"
285                             /\ channels' = [channels EXCEPT ![(out[
    self])] = @ (+) d_[self]]
286                             /\ pc' = [pc EXCEPT ![self] = "TP_Loop
    "]
287                             /\ UNCHANGED << flowin, flowout, in,
    out, d_, seen,
288                                              syncing, ended,
    outofsync, flowseq,
289                                              d_c, client_seq, d,
    server_seq >>
290
291 TP_flowsync_handle(self) == /\ pc[self] = "TP_flowsync_handle"
292                             /\ IF (d_[self].seq /= flowseq[self])
    /\ ~ended[self]
293                                   THEN /\ outofsync' = [outofsync
    EXCEPT ![self] = TRUE]
294                                   ELSE /\ TRUE
295                                        /\ UNCHANGED outofsync
296                             /\ IF syncing[self] /\ d_[self].msg = "
    UPDATE"
297                                   THEN /\ flowseq' = [flowseq
    EXCEPT ![self] = flowseq[self] + 1]
298                                        /\ UNCHANGED << channels,
    syncing,
299                                                        ended >>
300                                   ELSE /\ IF seen[self] /\ d_[self
    ].msg = "SEEN"
301                                              THEN /\ syncing' = [
    syncing EXCEPT ![self] = TRUE]
302                                                   /\ channels' = [
    channels EXCEPT ![(flowout[self])] = @ (+) ([msg |-> "UPDATE",
    seq |-> flowseq[self]])]
303                                                   /\ flowseq' = [
    flowseq EXCEPT ![self] = flowseq[self] + 1]
304                                                   /\ ended' = ended
305                                              ELSE /\ IF d_[self].
    msg = "ENDED"
306                                                        THEN /\
    ended' = [ended EXCEPT ![self] = TRUE]
```

```
307                                                                    /\
   flowseq' = [flowseq EXCEPT ![self] = flowseq[self] + 1]
308                                                                 ELSE /\
   TRUE
309                                                                    /\
   UNCHANGED << ended,
310
             flowseq >>
311                                                          /\ UNCHANGED <<
   channels,
312
   syncing >>
313                             /\ pc' = [pc EXCEPT ![self] = "TP_Loop
   "]
314                             /\ UNCHANGED << flowin, flowout, in,
   out, d_, seen,
315                                             d_c, client_seq, d,
   server_seq >>
316
317 TP(self) == TP_Loop(self) \/ TP_traffic_handle(self)
318                \/ TP_traffic_forward(self) \/ TP_flowsync_handle(
   self)
319
320 Client_sSYN(self) == /\ pc[self] = "Client_sSYN"
321                      /\ channels' = [channels EXCEPT ![(out[self])]
    = @ (+) ([seq |-> client_seq[self], msg |-> "SYN"])]
322                      /\ client_seq' = [client_seq EXCEPT ![self] =
   client_seq[self] + 1]
323                      /\ pc' = [pc EXCEPT ![self] = "Client_rSYNACK
   "]
324                      /\ UNCHANGED << flowin, flowout, in, out, d_,
   seen,
325                                      syncing, ended, outofsync,
   flowseq, d_c,
326                                      d, server_seq >>
327
328 Client_rSYNACK(self) == /\ pc[self] = "Client_rSYNACK"
329                         /\ channels[(in[self])] /= <<>>
330                         /\ d_c' = [d_c EXCEPT ![self] = Head(
   channels[(in[self])])]
331                         /\ channels' = [channels EXCEPT ![(in[self
   ])] = Tail(channels[(in[self])])]
332                         /\ Assert(d_c'[self].msg = "SYNACK",
333                                   "Failure of assertion at line
   113, column 5.")
334                         /\ pc' = [pc EXCEPT ![self] = "Client_sACK
   "]
335                         /\ UNCHANGED << flowin, flowout, in, out,
   d_, seen,
336                                         syncing, ended, outofsync,
   flowseq,
337                                         client_seq, d, server_seq
   >>
338
339 Client_sACK(self) == /\ pc[self] = "Client_sACK"
```

```
340                       /\ channels' = [channels EXCEPT ![(out[self])]
    = @ (+) ([seq |-> client_seq[self], msg |-> "ACK"])]
341                       /\ client_seq' = [client_seq EXCEPT ![self] =
   client_seq[self] + 1]
342                       /\ pc' = [pc EXCEPT ![self] = "Client_DataLoop
   "]
343                       /\ UNCHANGED << flowin, flowout, in, out, d_,
   seen,
344                                       syncing, ended, outofsync,
   flowseq, d_c,
345                                       d, server_seq >>
346
347 Client_DataLoop(self) == /\ pc[self] = "Client_DataLoop"
348                          /\ IF client_seq[self] < 14
349                                THEN /\ pc' = [pc EXCEPT ![self] = "
   Client_Data"]
350                                ELSE /\ pc' = [pc EXCEPT ![self] = "
   Client_sFIN"]
351                          /\ UNCHANGED << channels, flowin, flowout,
    in, out,
352                                          d_, seen, syncing, ended,
   outofsync,
353                                          flowseq, d_c, client_seq,
   d,
354                                          server_seq >>
355
356 Client_Data(self) == /\ pc[self] = "Client_Data"
357                      /\ \/ /\ channels[(in[self])] /= <<>>
358                            /\ d_c' = [d_c EXCEPT ![self] = Head(
   channels[(in[self])])]
359                            /\ channels' = [channels EXCEPT ![(in[
   self])] = Tail(channels[(in[self])])]
360                            /\ UNCHANGED client_seq
361                         \/ /\ channels' = [channels EXCEPT ![(out[
   self])] = @ (+) ([seq |-> client_seq[self], msg |-> "DATA"])]
362                            /\ client_seq' = [client_seq EXCEPT ![
   self] = client_seq[self] + 1]
363                            /\ d_c' = d_c
364                      /\ pc' = [pc EXCEPT ![self] = "Client_DataLoop
   "]
365                      /\ UNCHANGED << flowin, flowout, in, out, d_,
   seen,
366                                      syncing, ended, outofsync,
   flowseq, d,
367                                      server_seq >>
368
369 Client_sFIN(self) == /\ pc[self] = "Client_sFIN"
370                      /\ channels' = [channels EXCEPT ![(out[self])]
    = @ (+) ([seq |-> client_seq[self], msg |-> "FIN"])]
371                      /\ client_seq' = [client_seq EXCEPT ![self] =
   client_seq[self] + 1]
372                      /\ pc' = [pc EXCEPT ![self] = "Client_rFINACK
   "]
373                      /\ UNCHANGED << flowin, flowout, in, out, d_,
   seen,
```

```
374                                     syncing, ended, outofsync,
   flowseq, d_c,
375                                     d, server_seq >>
376
377 Client_rFINACK(self) == /\ pc[self] = "Client_rFINACK"
378                         /\ channels[(in[self])] /= <<>>
379                         /\ d_c' = [d_c EXCEPT ![self] = Head(
   channels[(in[self])])]
380                         /\ channels' = [channels EXCEPT ![(in[self
   ])] = Tail(channels[(in[self])])]
381                         /\ IF d_c'[self].msg = "ACK"
382                               THEN /\ pc' = [pc EXCEPT ![self] = "
   Client_rFIN"]
383                               ELSE /\ pc' = [pc EXCEPT ![self] = "
   Client_rFINACK"]
384                         /\ UNCHANGED << flowin, flowout, in, out,
   d_, seen,
385                                         syncing, ended, outofsync,
   flowseq,
386                                         client_seq, d, server_seq
   >>
387
388 Client_rFIN(self) == /\ pc[self] = "Client_rFIN"
389                      /\ channels[(in[self])] /= <<>>
390                      /\ d_c' = [d_c EXCEPT ![self] = Head(channels
   [(in[self])])]
391                      /\ channels' = [channels EXCEPT ![(in[self])]
   = Tail(channels[(in[self])])]
392                      /\ Assert(d_c'[self].msg = "FIN",
393                                "Failure of assertion at line 144,
   column 5.")
394                      /\ pc' = [pc EXCEPT ![self] = "Client_sFINACK
   "]
395                      /\ UNCHANGED << flowin, flowout, in, out, d_,
   seen,
396                                      syncing, ended, outofsync,
   flowseq,
397                                      client_seq, d, server_seq >>
398
399 Client_sFINACK(self) == /\ pc[self] = "Client_sFINACK"
400                         /\ channels' = [channels EXCEPT ![(out[self
   ])] = @ (+) ([seq |-> client_seq[self], msg |-> "ACK"])]
401                         /\ client_seq' = [client_seq EXCEPT ![self]
    = client_seq[self] + 1]
402                         /\ pc' = [pc EXCEPT ![self] = "Done"]
403                         /\ UNCHANGED << flowin, flowout, in, out,
   d_, seen,
404                                         syncing, ended, outofsync,
   flowseq,
405                                         d_c, d, server_seq >>
406
407 client(self) == Client_sSYN(self) \/ Client_rSYNACK(self)
408                 \/ Client_sACK(self) \/ Client_DataLoop(self)
409                 \/ Client_Data(self) \/ Client_sFIN(self)
410                 \/ Client_rFINACK(self) \/ Client_rFIN(self)
411                 \/ Client_sFINACK(self)
```

```
412
413 Server_rSYN(self) == /\ pc[self] = "Server_rSYN"
414                      /\ channels[(in[self])] /= <<>>
415                      /\ d' = [d EXCEPT ![self] = Head(channels[(in[
    self])])]
416                      /\ channels' = [channels EXCEPT ![(in[self])]
    = Tail(channels[(in[self])])]
417                      /\ Assert(d'[self].msg = "SYN",
418                                "Failure of assertion at line 159,
    column 5.")
419                      /\ pc' = [pc EXCEPT ![self] = "Server_sSYNACK
    "]
420                      /\ UNCHANGED << flowin, flowout, in, out, d_,
    seen,
421                                      syncing, ended, outofsync,
    flowseq, d_c,
422                                      client_seq, server_seq >>
423
424 Server_sSYNACK(self) == /\ pc[self] = "Server_sSYNACK"
425                         /\ channels' = [channels EXCEPT ![(out[self
    ])] = @ (+) ([seq |-> server_seq[self], msg |-> "SYNACK"])]
426                         /\ server_seq' = [server_seq EXCEPT ![self]
     = server_seq[self] + 1]
427                         /\ pc' = [pc EXCEPT ![self] = "Server_rACK
    "]
428                         /\ UNCHANGED << flowin, flowout, in, out,
    d_, seen,
429                                         syncing, ended, outofsync,
    flowseq,
430                                         d_c, client_seq, d >>
431
432 Server_rACK(self) == /\ pc[self] = "Server_rACK"
433                      /\ channels[(in[self])] /= <<>>
434                      /\ d' = [d EXCEPT ![self] = Head(channels[(in[
    self])])]
435                      /\ channels' = [channels EXCEPT ![(in[self])]
    = Tail(channels[(in[self])])]
436                      /\ Assert(d'[self].msg = "ACK",
437                                "Failure of assertion at line 167,
    column 5.")
438                      /\ pc' = [pc EXCEPT ![self] = "
    Server_Data_Loop"]
439                      /\ UNCHANGED << flowin, flowout, in, out, d_,
    seen,
440                                      syncing, ended, outofsync,
    flowseq, d_c,
441                                      client_seq, server_seq >>
442
443 Server_Data_Loop(self) == /\ pc[self] = "Server_Data_Loop"
444                           /\ IF d[self].msg /= "FIN"
445                                 THEN /\ pc' = [pc EXCEPT ![self] =
    "Server_Data"]
446                                 ELSE /\ pc' = [pc EXCEPT ![self] =
    "Server_sFINACK"]
447                           /\ UNCHANGED << channels, flowin, flowout
    , in, out,
```

```
448                                          d_, seen, syncing, ended,
    outofsync,
449                                          flowseq, d_c, client_seq,
    d,
450                                          server_seq >>
451
452 Server_Data(self) == /\ pc[self] = "Server_Data"
453                       /\ \/ /\ channels[(in[self])] /= <<>>
454                             /\ d' = [d EXCEPT ![self] = Head(
    channels[(in[self])])]
455                             /\ channels' = [channels EXCEPT ![(in[
    self])] = Tail(channels[(in[self])])]
456                             /\ UNCHANGED server_seq
457                          \/ /\ server_seq[self] < 24
458                             /\ channels' = [channels EXCEPT ![(out[
    self])] = @ (+) ([seq |-> server_seq[self], msg |-> "DATA"])]
459                             /\ server_seq' = [server_seq EXCEPT ![
    self] = server_seq[self] + 1]
460                             /\ d' = d
461                       /\ pc' = [pc EXCEPT ![self] = "
    Server_Data_Loop"]
462                       /\ UNCHANGED << flowin, flowout, in, out, d_,
    seen,
463                                       syncing, ended, outofsync,
    flowseq, d_c,
464                                       client_seq >>
465
466 Server_sFINACK(self) == /\ pc[self] = "Server_sFINACK"
467                         /\ channels' = [channels EXCEPT ![(out[self
    ])] = @ (+) ([seq |-> server_seq[self], msg |-> "ACK"])]
468                         /\ server_seq' = [server_seq EXCEPT ![self]
     = server_seq[self] + 1]
469                         /\ pc' = [pc EXCEPT ![self] = "Server_sFIN
    "]
470                         /\ UNCHANGED << flowin, flowout, in, out,
    d_, seen,
471                                         syncing, ended, outofsync,
    flowseq,
472                                         d_c, client_seq, d >>
473
474 Server_sFIN(self) == /\ pc[self] = "Server_sFIN"
475                      /\ channels' = [channels EXCEPT ![(out[self])]
     = @ (+) ([seq |-> server_seq[self], msg |-> "FIN"])]
476                      /\ server_seq' = [server_seq EXCEPT ![self] =
    server_seq[self] + 1]
477                      /\ pc' = [pc EXCEPT ![self] = "Server_rFINACK
    "]
478                      /\ UNCHANGED << flowin, flowout, in, out, d_,
    seen,
479                                      syncing, ended, outofsync,
    flowseq, d_c,
480                                      client_seq, d >>
481
482 Server_rFINACK(self) == /\ pc[self] = "Server_rFINACK"
483                         /\ channels[(in[self])] /= <<>>
```

```
                        /\ d' = [d EXCEPT ![self] = Head(channels[(
    in[self])])]
                        /\ channels' = [channels EXCEPT ![(in[self
    ])] = Tail(channels[(in[self])])]
                        /\ Assert(d'[self].msg = "ACK",
                                  "Failure of assertion at line
    191, column 5.")
                        /\ pc' = [pc EXCEPT ![self] = "Done"]
                        /\ UNCHANGED << flowin, flowout, in, out,
    d_, seen,
                                        syncing, ended, outofsync,
    flowseq,
                                        d_c, client_seq, server_seq
     >>

server(self) == Server_rSYN(self) \/ Server_sSYNACK(self)
                   \/ Server_rACK(self) \/ Server_Data_Loop(self)
                   \/ Server_Data(self) \/ Server_sFINACK(self)
                   \/ Server_sFIN(self) \/ Server_rFINACK(self)

Next == (\E self \in TPs: TP(self))
           \/ (\E self \in Clients: client(self))
           \/ (\E self \in Servers: server(self))

Spec == /\ Init /\ [][Next]_vars
        /\ WF_vars(Next)

\* END TRANSLATION

WillSync == <> \A tp \in TPs : syncing[tp] = TRUE

NeverDesync == [] \A tp \in TPs : outofsync[tp] = FALSE

FlowsyncPriority == [] \A tp \in TPs : (channels[flowin[tp]] /=
    <<>>) => (pc[tp] \notin {"TP_traffic_handle", "
    TP_traffic_forward"})

NeverDesyncWithFlowsyncPriority == FlowsyncPriority => NeverDesync

=============================================================================

\* Modification History
\* Last modified Wed Mar 01 13:49:43 CET 2023 by fbergman
\* Created Tue Feb 28 09:22:41 CET 2023 by fbergman
```

D

Classifier sharing and statistics synchronization model

```
/* Model parameters { */
#define N_REQUEST 2
#define ENABLE_CSTATE_2 true
/* } */

#define N_CONN 1

mtype = {
    /* TCP */
    SYN, SYNACK, ACK, HTTP_GET, HTTP_GET_MAL, HTTP_RSP,
   HTTP_RSP_MAL, FIN,
    /* TCP States */
    TCP_INIT, TCP_RUNNING, TCP_CLOSING, TCP_CLOSED,
    /* Stats to Database */
    STATS_FULL, STATS_HALF, STATS_RESET,
    /* FlowSync */
    SEEN, UPDATE, FORGET, ENDED,
    /* FlowSync Statistics */
    STATS_UPDATE, UPDATE_ACK, ACK_REQUEST, REJECT, FINAL_UPDATE,
   FINAL_ACK, REQUEST,
    RESPONSE, RESPONSE_ACK,
    /* States */
    FS_INIT, FS_SEEN, FS_SYNCING, FS_FALLBACK, FS_ACK_WAIT_TIMER,
    FS_LAST_ACK_TIMER, FS_WAIT_FOR_FINAL_UPDATE,
   FS_WAIT_FOR_FINAL_ACK,
    FS_HANDSHAKE_TIMER
};

typedef Stats {
    int up;
    int down;
}

typedef ConnectionInfo {
    mtype state = FS_INIT;
    bool out_of_sync = false;
    bool not_seen = false;
    bool upstream = false;

    int flow_seq = 0;
```

```
38      int classifier = 0;
39
40      Stats stats;
41      Stats peer_stats;
42      Stats synced_stats;
43      bool stats_dirty = false;
44  };
45
46  Stats db_full_stats, db_half_stats;
47  Stats real_stats;
48
49  chan FlowSync[2] = [1] of {int, mtype, int, int, Stats};
50  chan FlowSyncTxBuf[2] = [10] of {int, mtype, int, int, Stats};
51  chan Traffic[4] = [10] of {int, mtype};
52  chan TrafficTxBuf[2] = [10] of {int, mtype};
53  chan StatsChan = [100] of {mtype, Stats};
54
55  proctype Database() {
56      xr StatsChan;
57
58      Stats stats;
59  continue:
60      atomic {
61      if
62          :: StatsChan ? STATS_HALF, stats;
63              if
64                  :: stats.up > 0; db_half_stats.up = stats.up;
65                  :: else; skip;
66              fi
67              if
68                  :: stats.down > 0; db_half_stats.down = stats.down;
69                  :: else; skip;
70              fi
71              printf("DB half stats (up=%d, down=%d)\n",
    db_half_stats.up, db_half_stats.down);
72          :: StatsChan ? STATS_FULL, stats;
73              db_full_stats.up = stats.up;
74              db_full_stats.down = stats.down;
75              printf("DB full stats (up=%d, down=%d)\n",
    db_full_stats.up, db_full_stats.down);
76          :: StatsChan ? STATS_RESET, stats;
77              db_full_stats.up = 0;
78              db_full_stats.down = 0;
79              db_half_stats.up = 0;
80              db_half_stats.down = 0;
81              printf("Stats reset\n");
82      fi
83      goto continue;
84      }
85  }
86
87  proctype Tx(chan FlowIn, FlowOut, In, Out) {
88      xr FlowIn;
89      xs FlowOut;
90
91      xr In;
```

```
92      xs Out;
93
94      int a; mtype b; int c; int d; Stats e;
95      do
96          :: atomic{len(FlowIn) == 0 && len(FlowOut) == 0 && len(In)
    > 0;
97              In ? a, b;
98              Out ! a, b;}
99          :: atomic{FlowIn ? a, b, c, d, e;
100             if
101                 :: true; FlowOut ! a, b, c, d, e;
102                 :: b == STATS_UPDATE || b == UPDATE_ACK || b ==
    FINAL_ACK || b == FINAL_UPDATE;
103                     printf("Dropping %e\n", b);
104                     skip;
105             fi
106             }
107     od
108 }
109
110 proctype TP(chan FlowIn, FlowOut, Upstream, Downstream) {
111     //xr FlowIn;
112     //xs FlowOut;
113
114     xr Upstream;
115     //xs Downstream;
116
117     /* Only allow handshake timeout once per data packet */
118     bool can_handshake_timeout = false;
119
120     bool out_of_sync = false;
121     ConnectionInfo conn[N_CONN];
122
123     Stats stats_in;
124     Stats stats_to_db;
125
126     int conn_id; mtype type; int seq; int classifier;
127
128     int classifier_next;
129
130     do
131         // Handle Traffic packet
132         :: atomic {Upstream ? conn_id, type;
133             d_step {
134
135             can_handshake_timeout = true;
136
137             printf("Saw traffic: conn_id = %d, type = %e, flow_seq
    = %d\n", conn_id, type, conn[conn_id].flow_seq);
138
139             classifier_next = conn[conn_id].classifier;
140
141             if
142                 :: type == SYN;
143                     conn[conn_id].upstream = true;
144                     conn[conn_id].state = FS_SEEN;
```

```
145                     conn[conn_id].flow_seq = 0;
146                     conn[conn_id].classifier = 0;
147                     classifier_next = 0;
148                     conn[conn_id].stats.up = 0;
149                     conn[conn_id].stats.down = 0;
150                     conn[conn_id].peer_stats.up = 0;
151                     conn[conn_id].peer_stats.down = 0;
152                     conn[conn_id].synced_stats.up = 0;
153                     conn[conn_id].synced_stats.down = 0;
154                     conn[conn_id].stats_dirty = false;
155                     StatsChan ! STATS_RESET, conn[conn_id].stats;
156 
157                 :: type == SYNACK;
158                     printf("Syncing\n");
159                     conn[conn_id].state = FS_SYNCING;
160                     conn[conn_id].upstream = false;
161                     conn[conn_id].flow_seq = 0;
162                     conn[conn_id].classifier = 0;
163                     classifier_next = 0;
164                     conn[conn_id].stats.up = 0;
165                     conn[conn_id].stats.down = 0;
166                     conn[conn_id].peer_stats.up = 0;
167                     conn[conn_id].peer_stats.down = 0;
168                     conn[conn_id].synced_stats.up = 0;
169                     conn[conn_id].synced_stats.down = 0;
170                     conn[conn_id].stats_dirty = false;
171                     FlowOut ! conn_id, SEEN, 0, 0, conn[conn_id].
    stats;
172 
173                 :: type == ACK;
174                     FlowOut ! conn_id, UPDATE, conn[conn_id].
    flow_seq, conn[conn_id].classifier, conn[conn_id].stats;
175                     conn[conn_id].flow_seq++;
176 
177                 :: type == HTTP_GET && conn[conn_id].classifier ==
    0;
178                     classifier_next = 1;
179 
180                 :: type == HTTP_GET_MAL;
181                     classifier_next = 3;
182 
183                 :: type == HTTP_RSP && conn[conn_id].classifier ==
    1;
184                     if
185                         :: ENABLE_CSTATE_2;
186                             classifier_next = 2;
187                         :: else;
188                             skip;
189                     fi
190 
191                 :: type == HTTP_RSP_MAL && conn[conn_id].classifier
     == 3;
192                     classifier_next = 4;
193 
194                 :: type == FIN && conn[conn_id].state != FS_INIT;
195                     if
```

```
196                         :: conn[conn_id].upstream;
197                             conn[conn_id].state =
    FS_WAIT_FOR_FINAL_UPDATE;
198                             FlowOut ! conn_id, REQUEST, 0, conn[
    conn_id].classifier, conn[conn_id].stats;
199                         :: else;
200                             conn[conn_id].state =
    FS_WAIT_FOR_FINAL_ACK;
201                             FlowOut ! conn_id, FINAL_UPDATE, 0,
    conn[conn_id].classifier, conn[conn_id].stats;
202                     fi
203                     FlowOut ! conn_id, UPDATE, conn[conn_id].
    flow_seq, conn[conn_id].classifier, conn[conn_id].stats;
204                     conn[conn_id].flow_seq++;
205
206                 :: else; skip;
207             fi
208
209             // Only send UPDATE on classifier change
210             if
211                 :: classifier_next != conn[conn_id].classifier;
212                     conn[conn_id].classifier = classifier_next;
213                     FlowOut ! conn_id, UPDATE, conn[conn_id].
    flow_seq, conn[conn_id].classifier, conn[conn_id].stats;
214                     conn[conn_id].flow_seq++;
215                 :: else; skip;
216             fi
217
218             // Statistics counting
219             if
220                 :: type != ACK && type != FIN;
221                     if
222                         :: conn[conn_id].upstream;
223                             conn[conn_id].stats.up = conn[conn_id].
    stats.up + 1;
224                             if
225                                 :: conn[conn_id].state ==
    FS_FALLBACK;
226                                     stats_to_db.up = conn[conn_id].
    stats.up - conn[conn_id].peer_stats.up;
227                                     stats_to_db.down = 0;
228                                     StatsChan ! STATS_HALF,
    stats_to_db;
229                                 :: else;
230                                     StatsChan ! STATS_FULL, conn[
    conn_id].stats;
231                                     conn[conn_id].peer_stats.up =
    conn[conn_id].stats.up;
232                                     conn[conn_id].peer_stats.down =
     conn[conn_id].stats.down;
233                             fi
234                         :: else;
235                             conn[conn_id].stats.down = conn[conn_id
    ].stats.down + 1;
236                             if
```

```
237                                 :: conn[conn_id].state ==
    FS_FALLBACK;
238                                     stats_to_db.up = 0;
239                                     stats_to_db.down = conn[conn_id
    ].stats.down - conn[conn_id].synced_stats.down;
240                                     StatsChan ! STATS_HALF,
    stats_to_db;
241                                 :: else; skip;
242                             fi
243                     fi
244                     conn[conn_id].stats_dirty = true;
245                 :: else; skip;
246             fi
247
248             // Drop all traffic if we reach classifier >= 3
249             if
250                 :: conn[conn_id].classifier < 4;
251                     Downstream ! conn_id, type;
252                 :: else; printf("Dropping %e\n", type);
253             fi
254             }}
255
256         // Handle FlowSync packet.
257         :: atomic {FlowIn ? conn_id, type, seq, classifier,
    stats_in;
258             d_step {
259             printf("Saw flowsync: type = %e, seq = %d, classifier =
     %d\n", type, seq, classifier);
260
261             if
262                 :: conn[conn_id].state == FS_FALLBACK;
263                     if
264                         :: type == REJECT;
265                             skip;
266                         :: type == UPDATE || type == ENDED;
267                             skip;
268                         :: else;
269                             FlowOut ! conn_id, REJECT, 0, conn[
    conn_id].classifier, conn[conn_id].stats;
270                             goto continue;
271                     fi
272
273                 :: else; skip;
274             fi
275
276             if
277                 :: seq != conn[conn_id].flow_seq && seq != 0 &&
    conn[conn_id].state != FS_INIT;
278                     printf("Out of sync: seq = %d, flow_seq = %d,
    state = %e\n", seq, conn[conn_id].flow_seq, conn[conn_id].state)
    ;
279                     conn[conn_id].out_of_sync = true;
280                     out_of_sync = true;
281                 :: else; skip;
282             fi
283
```

```
284             if
285                 :: out_of_sync;
286                     FlowOut ! conn_id, FORGET, 0, conn[conn_id].
    classifier, conn[conn_id].stats;
287                     goto continue;
288                 :: else; skip;
289             fi
290
291             if
292                 :: conn[conn_id].state != FS_INIT && type == UPDATE
    ;
293                     conn[conn_id].classifier = classifier;
294                     conn[conn_id].flow_seq++;
295                 :: conn[conn_id].state == FS_SEEN && type == SEEN;
296                     printf("Syncing\n");
297                     conn[conn_id].state = FS_SYNCING;
298                     FlowOut ! conn_id, UPDATE, conn[conn_id].
    flow_seq, conn[conn_id].classifier, conn[conn_id].stats;
299                     conn[conn_id].flow_seq++;
300                 :: type == FORGET;
301                     conn[conn_id].state = FS_INIT;
302                     conn[conn_id].flow_seq++;
303                 :: type == ENDED;
304                     conn[conn_id].state = FS_INIT;
305                     conn[conn_id].flow_seq++;
306                 :: type == STATS_UPDATE;
307                     if
308                         :: conn[conn_id].upstream && conn[conn_id].
    state != FS_FALLBACK;
309                             conn[conn_id].stats.down = stats_in.
    down;
310                             FlowOut ! conn_id, UPDATE_ACK, 0, 0,
    conn[conn_id].stats;
311                         :: else; skip;
312                     fi
313                 :: type == UPDATE_ACK;
314                     conn[conn_id].synced_stats.up = stats_in.up;
315                     conn[conn_id].synced_stats.down = stats_in.down
    ;
316                     if
317                         :: conn[conn_id].state == FS_ACK_WAIT_TIMER
     || conn[conn_id].state == FS_LAST_ACK_TIMER;
318                             conn[conn_id].state = FS_SYNCING;
319                         :: else; skip;
320                     fi
321                 :: type == ACK_REQUEST;
322                     FlowOut ! conn_id, UPDATE_ACK, 0, 0, conn[
    conn_id].stats;
323                 :: type == FINAL_UPDATE;
324                     FlowOut ! conn_id, FINAL_ACK, 0, 0, conn[
    conn_id].stats;
325                     FlowOut ! conn_id, ENDED, 0, 0, conn[conn_id].
    stats;
326                     conn[conn_id].stats.down = stats_in.down;
327                     StatsChan ! STATS_FULL, conn[conn_id].stats;
328                     conn[conn_id].state = FS_INIT;
```

```
329                 :: type == FINAL_ACK;
330                     conn[conn_id].synced_stats.up = stats_in.up;
331                     conn[conn_id].synced_stats.down = stats_in.down
    ;
332                     conn[conn_id].state = FS_INIT;
333                 :: type == REQUEST;
334                     FlowOut ! conn_id, RESPONSE, 0, 0, conn[conn_id
    ].stats;
335                     conn[conn_id].state = FS_WAIT_FOR_FINAL_ACK;
336                 :: type == RESPONSE;
337                     FlowOut ! conn_id, RESPONSE_ACK, 0, 0, conn[
    conn_id].stats;
338                     FlowOut ! conn_id, ENDED, 0, 0, conn[conn_id].
    stats;
339                     conn[conn_id].stats.down = stats_in.down;
340                     StatsChan ! STATS_FULL, conn[conn_id].stats;
341                     conn[conn_id].state = FS_INIT;
342                 :: type == RESPONSE_ACK;
343                     conn[conn_id].synced_stats.up = stats_in.up;
344                     conn[conn_id].synced_stats.down = stats_in.down
    ;
345                     conn[conn_id].state = FS_INIT;
346                 :: type == REJECT;
347                     conn[conn_id].state = FS_FALLBACK;
348                 :: else;
349                     conn[conn_id].not_seen = true;
350             fi
351 continue:
352             skip;
353             }}
354         // Handshake timeout can occur at any point during sync (
    between data packets)
355         :: atomic{
356             can_handshake_timeout && timeout;
357             can_handshake_timeout = false;
358             int i;
359             for (i : 0 .. N_CONN - 1 ) {
360                 if
361                     :: conn[i].state == FS_SYNCING;
362                         if
363                             :: conn[i].upstream;
364                                 skip;
365                                 // We should send an UPDATE_ACK
    here, but that will fill the flowsync queue...
366                                 //FlowOut ! UPDATE_ACK, 0, stats;
367                             :: else;
368                                 printf("Ack wait timer 1 armed\n");
369                                 conn[i].state = FS_ACK_WAIT_TIMER;
370                                 FlowOut ! i, ACK_REQUEST, 0, 0,
    conn[conn_id].stats;
371                         fi
372                     // Maybe we just skip
373                     :: else; skip;
374                 fi
375             }
376             }
```

```
377         :: atomic {
378             timeout;
379             //len(Upstream) == 0 && len(FlowIn) == 0;
380             int i;
381             for (i : 0 .. N_CONN - 1 ) {
382                 if
383                     :: conn[i].state == FS_ACK_WAIT_TIMER;
384                         printf("Ack wait timer 1 timed out\n");
385                         conn[i].state = FS_LAST_ACK_TIMER;
386                         FlowOut ! i, ACK_REQUEST, 0, 0, conn[
    conn_id].stats;
387                     :: conn[i].state == FS_LAST_ACK_TIMER;
388                         printf("Ack wait timer 2 timed out\n");
389                         conn[i].state = FS_FALLBACK;
390                         FlowOut ! i, REJECT, 0, 0, conn[conn_id].
    stats;
391                     :: conn[i].state == FS_WAIT_FOR_FINAL_ACK;
392                         printf("Final ack wait timer timed out\n");
393                         stats_to_db.up = 0;
394                         stats_to_db.down = conn[conn_id].stats.down
     - conn[conn_id].synced_stats.down;
395                         StatsChan ! STATS_HALF, stats_to_db;
396                         FlowOut ! i, ENDED, 0, 0, conn[conn_id].
    stats;
397                         conn[i].state = FS_INIT;
398                     :: conn[i].stats_dirty;
399                         FlowOut ! conn_id, STATS_UPDATE, 0, 0, conn
    [conn_id].stats;
400                         conn[i].stats_dirty = false;
401                     // Maybe we just skip
402                     :: else; skip;
403                 fi
404             }
405             }
406     od
407 }
408
409 proctype Server(chan In, Out) {
410     xr In;
411     //xs Out;
412
413     mtype state[N_CONN];
414
415     int conn_id; mtype type;
416
417     do
418         // Receive packets.
419         :: atomic{In ? conn_id, type;
420             printf("Server Received %e\n", type);
421             if
422                 :: type == SYN;
423                     state[conn_id] = TCP_INIT;
424                     real_stats.down = real_stats.down + 1;
425                     Out ! conn_id, SYNACK;
426                 :: type == ACK && state[conn_id] == TCP_INIT;
427                     state[conn_id] = TCP_RUNNING;
```

```
428                 :: type == HTTP_GET && state[conn_id] ==
    TCP_RUNNING;
429                     real_stats.down = real_stats.down + 1;
430                     Out ! conn_id, HTTP_RSP;
431                 :: type == HTTP_GET_MAL && state[conn_id] ==
    TCP_RUNNING;
432                     real_stats.down = real_stats.down + 1;
433                     Out ! conn_id, HTTP_RSP_MAL;
434                 :: type == FIN;
435                     Out ! conn_id, ACK;
436                     Out ! conn_id, FIN;
437                 :: type == ACK && state[conn_id] == TCP_CLOSING;
438                     state[conn_id] = TCP_CLOSED;
439                 :: else; skip;
440             fi
441             }
442     od
443 }
444
445 proctype Client(chan In, Out; int conn_id) {
446     bool sent_mal = false;
447     bool received_mal = false;
448     mtype type;
449     int count = N_REQUEST;
450     int received = 0;
451
452 start:
453     real_stats.up = 0;
454     real_stats.down = 0;
455
456     // TCP handshake.
457     timeout;
458 atomic{
459     Out ! conn_id, SYN;
460     real_stats.up = real_stats.up + 1;
461     In ?? eval(conn_id), SYNACK;
462     Out ! conn_id, ACK;
463
464     count = N_REQUEST;
465     received = 0;
466 }
467
468 atomic{
469     do
470         :: count > 0 && timeout;
471             if
472                 :: Out ! conn_id, HTTP_GET;
473                 :: Out ! conn_id, HTTP_GET_MAL;
474                     sent_mal = true;
475             fi
476             real_stats.up = real_stats.up + 1;
477             count = count - 1;
478         ::  In ?? eval(conn_id), type;
479             if
480                 :: type == HTTP_RSP_MAL;
481                     received_mal = true;
```

```
482                 :: else; skip;
483             fi
484             received = received + 1;
485
486             printf("Client Received %e\n", type);
487         :: received == N_REQUEST;
488             break;
489     od
490 }
491
492 atomic{
493     // Termination handshake.
494     timeout;
495     Out ! conn_id, FIN;
496
497     // Wait for ACK.
498     do
499         :: In ?? eval(conn_id), type;
500             if
501                 :: type == ACK;
502                     break;
503                 :: else; skip;
504             fi
505     od
506
507     In ?? eval(conn_id), FIN;
508     timeout;
509     Out ! conn_id, ACK;
510
511     printf("Client done.\n");
512 }
513 done:
514     goto start;
515 }
516
517 init {
518 atomic {
519     run TP(FlowSync[0], FlowSyncTxBuf[0], Traffic[0], Traffic[2]);
520     run TP(FlowSync[1], FlowSyncTxBuf[1], Traffic[1], Traffic[3]);
521
522     run Server(Traffic[2], TrafficTxBuf[1]);
523     int i;
524     for (i : 0 .. N_CONN - 1) {
525         run Client(Traffic[3], TrafficTxBuf[0], i);
526     }
527
528     run Tx(FlowSyncTxBuf[0], FlowSync[1], TrafficTxBuf[1], Traffic
    [1]);
529     run Tx(FlowSyncTxBuf[1], FlowSync[0], TrafficTxBuf[0], Traffic
    [0]);
530
531     run Database();
532 }
533 }
534
535 ltl forever_fin {
```

```
536     ([] ! Client[4]:sent_mal) -> ([] <> Client[4]@done)
537 }
538 ltl never_desync {
539     ([] ! (TP[1]:out_of_sync || TP[2]:out_of_sync))
540 }
541 ltl never_receive_mal {
542     ([] ! Client[4]:received_mal)
543 }
544 ltl max_down_stats {
545     [] (Client[4]@done -> db_full_stats.down + db_half_stats.down
        <= real_stats.down)
546 }
547 ltl max_up_stats {
548     [] (Client[4]@done -> db_full_stats.up + db_half_stats.up <=
        real_stats.up)
549 }
```

E

Link redundancy model in Promela

```
#define N_TP 3
#define N_DOMAIN 2
#define N_LINK_FAILURE 2
#define THREAD_0_PRIO 2

mtype = {HELLO, OHAI, SERVICE}

typedef Domain {
    bool activeTPList [N_TP] = false;
    int active_count=0;
    bool ok;
}

typedef Link {
    bool tp[N_TP];
}

//TP node that represents a TP in a domain.
typedef TPNode {
    Domain domain[N_DOMAIN];
}

typedef Message {
    mtype message;
    int domain;
    int src;
    int dst;
}

typedef SwitchDef {
    chan domainMsgsChan = [10] of {Message};
}

Link links[N_DOMAIN];

int l, activeCount1 = 0;

//inline code to calculate the active nodes when OHAI message is
   received.
inline countActiveTP (tpId, domainId) {
    d_step {
    activeCount1 = 0;
    for (l: 0 .. N_TP-1) {
```

```
43          if
44              ::  l != tpId && TPList[tpId].domain[domainId].
    activeTPList[l] ->
45                  activeCount1++;
46              ::  else ->
47                  skip;
48          fi
49      }
50
51      TPList[tpId].domain[domainId].active_count = activeCount1;
52      }
53  }
54
55  chan TPIn[N_TP] = [10] of {Message};
56
57
58  SwitchDef switchs[N_DOMAIN];
59  TPNode TPList[N_TP];
60
61  proctype Service(int tpId, dst) {
62      int i;
63      Message m;
64      m.src = tpId;
65      m.message = SERVICE;
66      m.dst = dst;
67
68  start:
69      if
70          // Send a message on any active domain
71          :: TPList[tpId].domain[0].ok;
72              switchs[0].domainMsgsChan ! m;
73          :: TPList[tpId].domain[1].ok;
74              switchs[1].domainMsgsChan ! m;
75      fi
76  sent:
77      goto start;
78  }
79
80  proctype Thread0(chan In; int tpId)
81  {
82      xr In;
83
84      Message hello;
85      Message ohai, incomingMsg;
86
87      d_step {
88          //These assignments are of no use in current model but they
    will be in the suggested one.
89          TPList[tpId].domain[0].activeTPList[tpId] = true;
90          TPList[tpId].domain[1].activeTPList[tpId] = true;
91
92          hello.message = HELLO;
93          hello.dst = 255;
94          hello.src = tpId;
95
96          ohai.message = OHAI;
```

```
97          ohai.src = tpId;
98      }
99      atomic{
100         int d;
101         for (d: 0 .. N_DOMAIN-1){
102             switchs[d].domainMsgsChan ! hello;
103         }
104     }
105
106     do
107         ::  In ? incomingMsg ->
108                 printf("Domain %d: %e TP%d -> TP%d\n", incomingMsg.
    domain, incomingMsg.message, incomingMsg.src, tpId);
109                 if
110                     ::  atomic{
111                         incomingMsg.message == HELLO;
112                         ohai.dst = incomingMsg.src;
113                         switchs[incomingMsg.domain].domainMsgsChan
    ! ohai;
114                         }
115                     ::  atomic{
116                         incomingMsg.message == OHAI;
117                         TPList[tpId].domain[incomingMsg.domain].
    activeTPList[incomingMsg.src] = true;
118                         countActiveTP(tpId, incomingMsg.domain);
119                         // Figure out which domains should be
    active
120                         int max_active = 0;
121                         int d;
122                         for (d : 0 .. N_DOMAIN-1) {
123                             if
124                                 :: TPList[tpId].domain[d].
    active_count > max_active -> max_active = TPList[tpId].domain[d
    ].active_count;
125                                 :: else; skip;
126                             fi
127                         }
128                         for (d : 0 .. N_DOMAIN-1) {
129                             TPList[tpId].domain[d].ok = TPList[tpId
    ].domain[d].active_count == max_active;
130                         }
131                         }
132                     ::  incomingMsg.message == SERVICE;
133 receive_service:
134                         true;
135                 fi
136     od
137 }
138
139 proctype Switch(chan fIn; int domainId) {
140     xr fIn;
141
142     int i;
143     Message messageIn;
144
145     do
```

```
146         ::  atomic{fIn ? messageIn ->
147             d_step {
148                 messageIn.domain = domainId;
149                 if
150                     :: messageIn.dst == 255 ->
151                         for (i : 0 .. N_TP-1) {
152                             if
153                                 ::  i != messageIn.src ->
154                                     if
155                                         ::  links[domainId].tp[
    i] ->
156                                             printf("Forwarding
    %e from TP%d to TP%d\n", messageIn.message, messageIn.src, i);
157                                             TPIn[i] ! messageIn
    ;
158                                         ::  else ->
159
160                                             printf("Dropping %e
     from TP%d to TP%d\n", messageIn.message, messageIn.src, i);
161                                     fi
162                                 ::  else ->
163                                     skip;
164                             fi
165                         }
166                     ::  else ->
167                         if
168                             :: links[domainId].tp[messageIn.dst] ->
     TPIn[messageIn.dst] ! messageIn;
169                             :: else ->
170                                 printf("Dropping %e from TP%d
    to TP%d\n", messageIn.message, messageIn.src, messageIn.dst);
171                         fi
172                 fi
173             }
174             }
175     od
176 }
177
178 init {
179 atomic {
180     // Set up links
181     int i, j, k;
182     for (i : 0 .. N_DOMAIN-1) {
183         for (j : 0 .. N_TP-1) {
184             links[i].tp[j] = true;
185         }
186     }
187
188     // Disable N_LINK_FAILURE links
189     for (k : 0 .. N_LINK_FAILURE-1) {
190         select (i : 0 .. N_DOMAIN-1);
191         select (j : 0 .. N_TP-1);
192         links[i].tp[j] = false;
193     }
194
195     run Switch(switchs[0].domainMsgsChan, 0) priority 3;
```

```
196     run Switch(switchs[1].domainMsgsChan, 1) priority 3;
197
198     run Thread0(TPIn[0], 0) priority THREAD_0_PRIO;
199     run Thread0(TPIn[1], 1) priority THREAD_0_PRIO;
200     run Thread0(TPIn[2], 2) priority THREAD_0_PRIO;
201
202     run Service(0, 1) priority 1;
203 }
204 }
205
206 ltl eventually_service {
207     [] (Service[6]@sent -> <> Thread0[3 + 1]@receive_service)
208 }
209
210 ltl domain_agreement {
211     <> [] (TPList[0].domain[0].ok == TPList[1].domain[0].ok
212         && TPList[1].domain[0].ok == TPList[2].domain[0].ok
213         && TPList[1].domain[1].ok == TPList[2].domain[1].ok
214         && TPList[0].domain[1].ok == TPList[1].domain[1].ok
215         )
216 }
```

F

Link redundancy error trail

```
ltl eventually_service: [] ((! ((Service[6]@sent))) || (<> ((Thread0[(3+1)]@receive_service))))
ltl domain_agreement: <> ([] (((((TPList[0].domain[0].ok==TPList[1].domain[0].ok)) && ((TPList[1].domain[0].ok==TPList[2].domain[0].ok))) && ((TPList[1].domain[1].ok==TPList[2].domain[1].ok))) && ((TPList[0].domain[1].ok==TPList[1].domain[1].ok))))
Never claim moves to line 4 [(1)]
               Forwarding HELLO from TP0 to TP1
               Forwarding HELLO from TP0 to TP2
                   Dropping HELLO from TP0 to TP1
                   Forwarding HELLO from TP0 to TP2
               Dropping HELLO from TP1 to TP0
               Forwarding HELLO from TP1 to TP2
                   Forwarding HELLO from TP1 to TP0
                   Forwarding HELLO from TP1 to TP2
                           Domain 0: HELLO TP0 -> TP1
               Dropping OHAI from TP1 to TP0
                        Domain 1: HELLO TP1 -> TP0
                   Dropping OHAI from TP0 to TP1
               Dropping HELLO from TP2 to TP0
               Forwarding HELLO from TP2 to TP1
                   Forwarding HELLO from TP2 to TP0
                   Dropping HELLO from TP2 to TP1
                                Domain 0: HELLO TP0 -> TP2
               Dropping OHAI from TP2 to TP0
                                Domain 1: HELLO TP0 -> TP2
                                Domain 0: HELLO TP1 -> TP2
                                Domain 1: HELLO TP1 -> TP2
                   Dropping OHAI from TP2 to TP1
                           Domain 0: HELLO TP2 -> TP1
                               Domain 0: OHAI TP1 -> TP2
                           Domain 0: OHAI TP2 -> TP1
                       Domain 1: HELLO TP2 -> TP0
                               Domain 1: OHAI TP0 -> TP2
                       Domain 1: OHAI TP2 -> TP0
Never claim moves to line 3 [((!(!((Service[6]._p==sent)))&&!((Thread0[(3+1)]._p==receive_service))))]
                   Dropping SERVICE from TP0 to TP1
Never claim moves to line 8 [(!((Thread0[(3+1)]._p==receive_service)))]
                   Dropping SERVICE from TP0 to TP1
                   Dropping SERVICE from TP0 to TP1
                   Dropping SERVICE from TP0 to TP1
```

```
                    Dropping SERVICE from TP0 to TP1
  <<<<<START OF CYCLE>>>>>
                    Dropping SERVICE from TP0 to TP1
                    Dropping SERVICE from TP0 to TP1
                    Dropping SERVICE from TP0 to TP1
                    Dropping SERVICE from TP0 to TP1
                    Dropping SERVICE from TP0 to TP1
spin: trail ends after 323 steps
#processes: 7
    links[0].tp[0] = 0
    links[0].tp[1] = 1
    links[0].tp[2] = 1
    links[1].tp[0] = 1
    links[1].tp[1] = 0
    links[1].tp[2] = 1
    l = 3
    activeCount1 = 1
    queue 3 (TPIn[0]):
    queue 4 (TPIn[1]):
    queue 5 (TPIn[2]):
    queue 1 (domainMsgsChan):
    queue 2 (domainMsgsChan): [SERVICE,0,0,1]
    TPList[0].domain[0].activeTPList[0] = 1
    TPList[0].domain[0].activeTPList[1] = 0
    TPList[0].domain[0].activeTPList[2] = 0
    TPList[0].domain[0].active_count = 0
    TPList[0].domain[0].ok = 0
    TPList[0].domain[1].activeTPList[0] = 1
    TPList[0].domain[1].activeTPList[1] = 0
    TPList[0].domain[1].activeTPList[2] = 1
    TPList[0].domain[1].active_count = 1
    TPList[0].domain[1].ok = 1
    TPList[1].domain[0].activeTPList[0] = 0
    TPList[1].domain[0].activeTPList[1] = 1
    TPList[1].domain[0].activeTPList[2] = 1
    TPList[1].domain[0].active_count = 1
    TPList[1].domain[0].ok = 1
    TPList[1].domain[1].activeTPList[0] = 0
    TPList[1].domain[1].activeTPList[1] = 1
    TPList[1].domain[1].activeTPList[2] = 0
    TPList[1].domain[1].active_count = 0
    TPList[1].domain[1].ok = 0
    TPList[2].domain[0].activeTPList[0] = 0
    TPList[2].domain[0].activeTPList[1] = 1
    TPList[2].domain[0].activeTPList[2] = 1
    TPList[2].domain[0].active_count = 1
    TPList[2].domain[0].ok = 1
    TPList[2].domain[1].activeTPList[0] = 1
    TPList[2].domain[1].activeTPList[1] = 0
    TPList[2].domain[1].activeTPList[2] = 1
    TPList[2].domain[1].active_count = 1
    TPList[2].domain[1].ok = 1
    sent = 0
    receive_service = 0
323:  proc  6 (Service:1) redundency.pml:100 (state 10)
323:  proc  5 (Thread0:2) redundency.pml:142 (state 76)
```

```
94 323:  proc  4 (Thread0:2) redundency.pml:142 (state 76)
95 323:  proc  3 (Thread0:2) redundency.pml:142 (state 76)
96 323:  proc  2 (Switch:3) redundency.pml:181 (state 35)
97 323:  proc  1 (Switch:3) redundency.pml:181 (state 35)
98 323:  proc  0 (:init::1) redundency.pml:240 (state 51) <valid end
      state>
99 323:  proc  - (eventually_service:1) _spin_nvr.tmp:7 (state 10)
100 7 processes created
```