Formalization and Verification of the Shim6 Protocol

Matthijs Mekking

May 2007

Contents

Acknowledgments **5**

1 Introduction **7**
1.1 Background . . . 7
1.2 Problem description . . . 9
1.2.1 Draft specification of Shim6 . . . 9
1.2.2 Implementation experience . . . 10
1.3 Outline . . . 11

2 Shim6 **13**
2.1 Architecture . . . 15
2.2 Protocol overview . . . 16
2.2.1 Context establishment exchange . . . 17
2.2.2 Failure detection and locator pair exploration . . . 21

3 Formal methods applied to Shim6 **23**
3.1 Modeling context establishment . . . 24
3.1.1 The network . . . 25
3.1.2 The context . . . 27
3.1.3 The dispatcher . . . 31
3.1.4 The upper layer protocol . . . 34
3.1.5 Initializing the model . . . 36
3.2 Adding REAP . . . 37
3.2.1 Failure detection and locator pair exploration . . . 38

4 Verification **45**
4.1 Abstractions . . . 46
4.2 Results . . . 48
4.2.1 Context establishment . . . 48
4.2.2 Reachability protocol . . . 50
4.3 Formalization results . . . 52

5 Conformance testing **55**
5.1 Test setting . . . 55
5.1.1 The shimulator . . . 55
5.1.2 The virtual network . . . 56
5.1.3 The Wireshark traffic analyzer . . . 57

5.2 Results 60
5.2.1 Implementation requirements 61
5.2.2 Test conclusions 69

6 Conclusions **71**
6.1 Future Work 72

Appendices

A List of abbreviations **73**

B Adjustments to the Shim6 model **77**
B.1 Updated context establishment 77
B.2 Abstracted context establishment 79
B.2.1 Packet structure 79
B.2.2 Merging locations i2sent and i2bissent 79
B.2.3 All Shim6 enabled hosts 80
B.2.4 No context forking 81

Acknowledgments

Foundation NLnet Labs

First of all, I would like to thank the foundation NLnet Labs. They provided me the opportunity to perform my final project on a subject that I was highly interested in. I appreciated the openness of the my research, as well as the technical support. This contributed to a very good work environment. I also enjoyed the technical meetings where I could share my results with the NLnet Labs team, but where I also could learn about other Internet protocols like, for example, Unbound and DNSSEC. I would especially like to thank Wouter Wijngaards, my supervisor at NLnet Labs. He supported me with the technical unclarities, helped me improving my thesis and guided the project extensively.

Readers

I would like to thank everybody that have been taking their time to read this thesis. Special thanks goes out to those readers that helped me improve my thesis scientifically and linguistically: my girlfriend Véronique Lensen, my friends Martijn Velthausz and Jelte Jansen, the latter also being a member of the NLnet Labs team, Olaf Kolkman and Wouter Wijngaards of the NLnet Labs foundation, Theo Schouten and Frits Vaandrager, my supervisors at the Radboud University.

Friends and family

Last but not least, I would like to thank my parents for supporting me financially. Together with my girlfriend, they helped me trough the difficult times and they showed special, non-scientifically interest in my project.

Chapter 1

Introduction

1.1 Background

People become more and more dependent on modern communication technology. Nowadays, the Internet is used for a broad variety of work and leisure activities and is accessed via more and more devices such as wireless computers and mobile phones. The more activities you have on the network, the more important the reliability of your network connection becomes. As an Internet user, you probably recognize the annoyance when using the Internet, the network connection suddenly fails and you are no longer able to continue your work. Also, if you provide a service on the Internet, whether you sell products or share your travel adventures with others, you would like to be reachable to the rest of the world as much as possible.

An increasingly used technique to enlarge the reliability of the network connection is called *multihoming*. The most important reasons for multihoming are [Abl05]:

- **Redundancy** In the case of a link failure, the connection may continue at a different, working link. Normally, a computer device is connected to the Internet with one single link. If this link fails, for example due to a fiber cut or a bad router interface, the connection will go down. When multihomed, a device is provided with multiple links, protecting you against these failures.
- **Load sharing** Load sharing refers to distributing incoming and outgoing traffic using different links, leading to more efficient and possibly faster communication.
- **Performance** With multihoming, a device can be protected from performance difficulties on its outgoing links. If there is a congestion on a certain path, outgoing messages can be transmitted using a different link, so that the congested path is avoided.
- **Policy** Another reason to multihome is choice. For example, provider X might offer traffic of a certain type at lower cost than provider Y does. By setting up policies, the device is able to route this type of packets via provider X and other packets via provider Y, resulting in a more advantageous situation for the user.

In the current approach, multihoming is deployed using routing. The Border Gateway Protocol (BGP-4, [Rek06]) is a routing information protocol and is used to announce routes to the customer from two or more service providers. This provides the rest of the Internet with multiple paths back to the multihomed site and the service provider supplies the multihomer with additional possible paths for its outgoing traffic. In cases of a failing outgoing link, outgoing traffic will automatically be routed via one of the remaining links. BGP-4 will notify other networks they need to route incoming traffic via another service provider and link.

Multihoming with BGP-4 requires IP address space. This is supplied by a Regional Internet Registry (RIR), an organization overseeing the allocation and registration of Internet number resources within a region. There are two types of address space that can be issued: provider assigned (PA) space and provider independent (PI) space. PA space is assigned to you by your primary service provider, while PI space is directly received from a RIR. For PI address space, certain requirements have to be satisfied. The main advantage of PI space is that when switching providers, you don't have to renumber your IP addresses, which is a complex process [Car96]. Furthermore, PA addresses are subject to being filtered. However, when the announcement is filtered out, packets still flow towards the primary service provider. But if this occurs on a large scale, the multihomer must rely upon its main provider. This reliance might result in an effective single point of failure.

So for multihoming, PI address space is preferred. If an organization has sufficient IP addressing requirements, it may be supplied with PI space. But many organizations that do wish to multihome cannot meet these requirements. Besides, there aren't enough IP addresses to provide multihoming to everybody. Another problem is that each site that uses PI addresses introduces an additional prefix into the global routing table. Being reachable through any of its providers implies that a customer network must be visible in the inter domain routing system. This will increase the number of prefixes in the routing tables and the route selection procedure will require more resources. The growth of the routing table is recognized as a scalability problem [Mey06].

The currently used Internet mainly uses version 4 of the Internet Protocol (IPv4) and does not provide enough addresses to multihome on a large scale. IPv4 addresses are represented by 32 bits, that is, there are 2^{32} different possible addresses. This address space is insufficient if the number of Internet access points grow considerably. Due to these limitations, the Internet Engineering Task Force (IETF) started to develop a successor protocol, called IPv6 [Dee98]. The most important changes are:

- **Extended address space** The address format is extended from 32 bits to 128 bits, providing plenty of IP addresses.
- **Stateless autoconfiguration** In IPv4, an unique IP address for every device would have to be assigned, using the Dynamic Host Configuration Protocol (DHCP) or through manual configuration. In IPv6, if a new booting device comes up and asks for its network prefix, it would get one or more prefixes from a router on its link. With this prefix information, it can autoconfigure one or more valid IP addresses.

- **Simplification of header format** Only the required information is stored in the header format. Optional information is stored in extension headers. The simpler header format allows faster processing and more flexibility in extending the protocol.

1.2 Problem description

To summarize, multihoming is a great technique for increasing connection reliability and traffic engineering purposes, but it also increases concerns about the scalability of address space and routing techniques [Cla91]. Shim6 is a multihoming protocol that is being developed in order to solve these issues.

1.2.1 Draft specification of Shim6

The most important and most often used Internet protocols are standardized by the IETF. Since these standards serve as a guide to many different developers, it is important that these documents only allow for one clear interpretation, are complete and ensure the required functionality. The IETF has defined a process to create Internet standards (RFC 2026, [Bra96]) in order to improve the specifications. These documents are often written in an informal language that allows ambiguities, omissions and inconsistencies that are difficult to detect.

At the time of writing, Shim6 is very close to become a proposed standard. This means that it has enough community interest to be considered valuable. The amount of work that is being done at this moment reflects the high rate of interest. Several organizations are currently developing Shim6 implementations (see [Bar06], [Tae06] and [Hen06]). A proposed standard usually does not require an implementation of the referenced protocol, because at this point of time, the document may still contain problems. Therefore, RFC 2026 suggests that developers should treat the proposed standard as immature specifications. Implementations made at this point should especially be used to gain experience and clarify specifications. It includes extensive testing and a thoroughly study of the source code. This method however has its disadvantages. Given an implementation, it still might be hard to detect some types of errors. Even if an error is detected, it can be difficult to solve this in the implementation. Also, implementing a protocol is a time-consuming activity. In short, this process takes quite some time, needs to be carried out by people with a lot of knowledge and it is still hard to detect flaws.

Applying formal methods can increase the reliability and robustness of a specification. Formal methods may help to detect issues, and help to improve the quality of the protocol standards. Van Langevelde, Romijn, and Goga [van03] for example, applied formal methods during the development of the IEEE 1394.1 FireWire Net Update standard. This work resulted in the discovery and correction of many errors, omissions and inconsistencies. The advantages with regard to experimental implementations are that it gives more provable certainty, flaws can be detected and fixed with more ease. Unfortunately, formal methods still require a lot of knowledge of the protocol as well as formal notations. Although the assessment of Shim6 within

the IETF is substantial, attempts to include formal descriptions for protocols only succeeded partly. Some protocol standard do include finite state machines (FSMs), but these are mostly illustrative and partly informal.

One type of verification is called model checking. Model checking allows automatic verification of certain system properties at a suitable level of abstraction. Van Langevelde et al. showed that model checking can be successfully applied to protocol descriptions or implementations, but the technique also has its practical limitations. The main problem is the so-called state space explosion problem. Modeling a system in realistic detail is practically impossible because we need to deal with all possible behavior of the environment of the system. This requires a great number of variables and processes, resulting in an exponential growth of the state space.

Model checkers cope with the state space explosion problem by using techniques such as abstraction, decomposition and symmetry reduction. Abstraction is used when a system does not depend on an actual value of a certain variable, but rather on a general value. For example, an action can depend on the value of a variable `i`. Suppose `i` is used in an `if` construction: `if(i<0)`, the impact of value of `i` can be reduced to two abstract values *positive* and *negative*. Decomposition refers to reducing a system into a smaller system. Parts of the system that do not influence a certain property, can be safely removed. Symmetry reduction is a method to detect identical states that can be mapped onto one state. This significantly reduces the state space. On the other hand, finding the right balance between abstraction and reality can be hard. The level of abstraction should be high enough to simplify the system and avoid state space explosion, but should not obscure the relationship between the model and the system that they represent.

Applying formal methods and constructing a prototype implementation are useful techniques in order to verify properties that a protocol must satisfy. Both techniques have its own advantages and disadvantages, but may complement each other while verifying a standard. Currently, the IETF only uses the latter technique. In this thesis, I shall show how formal methods can be applied to Shim6, in order to clarify the specifications. Such notation should be easy to understand, so that programmers are able to read it. It should have the right abstractions, so that the relationship with the document is clear, but does not suffer from the state space explosion problem. I shall provide formal notations that meet the suggested requirements. Such activity can be applied to other Internet standards as well, to improve the quality of network protocols.

I have modeled two critical phases of the Shim6 protocol. The two algorithms have been modeled with a model checker called Uppaal . The tool allows us to specify, validate and verify models of real-time systems. This revealed several errors that were not spotted before, and that were difficult to derive from the protocol specification.

1.2.2 Implementation experience

The IETF considers that implementation is a strong argument in favor of a proposed standard designation. Such process is not required, but is considered desirable

(the Internet Engineering Steering Group (IESG) may require implementation to a specification that affects the core Internet protocols, which is true for Shim6). The specification should also have no technical omissions. The problem here is that it is difficult to determine if the implementation in question is conform the specification. And because the documents are written in informal language, this process becomes even more difficult.

There have been defined some rules to facilitate the process (RFC 2119, [Bra97]). RFC 2119 defines keywords that indicate the requirements. The keywords can be included in the specification and are written in capitals. For example, the keyword *MUST* means that the definition is an absolute requirement, and *MUST NOT* means that the corresponding behavior is absolutely prohibited. There exists weaker forms of keywords like *SHOULD*, meaning that there may exist valid reasons to ignore the requirement, but the full implications must be understood and carefully weighed, or *MAY*, meaning the requirement is truly optional.

I have performed a conformance test on one of the Shim6 implementations. These implementation already have contributed to the Shim6 draft specification, but did not mentioned the problems that were found through model checking. By performing such a test, I try to find out if the implementation correctly follows the specification. If so, it should encounter the same problems as our formalized model. The test results can than form a basis for comparing the two methods in question. Because the implementation is a prototype version, several other problems were discovered.

1.3 Outline

Chapter 2 (page 13) provides a more detailed description of the Shim6 protocol. This section discusses the protocol being examined in this thesis.

Chapter 3 (page 23) shows how formal methods have been applied to the protocol description. With the use of model checking techniques I tried to construct a realistic formal model that is easy to understand by engineers, and may serve as a basis for formal verification.

Chapter 4 (page 45) describes the verification process applied to the formalized model. I will show which issues were encountered, the properties that have been verified, and the results of the verification process.

Chapter 5 (page 55) describes a conformance test of a Shim6 implementation. If this version of Shim6 is implemented conform the specification, it should suffer from the same issues as our formalized model. If so, this indicates that formal methods may reveal troubles that are easily overlooked and affect real implementations.

Chapter 6 (page 71) will summarize the conclusions I have drawn based upon the work described in this thesis. I will also discuss some possible future work on the subject of Shim6 verification and implementation.

Appendix A (page 73) provides a list of abbreviations used in this thesis.

Appendix B (page 77) shows some design issues for the Shim6 model. First of all, some modifications have been made in order to satisfy the properties. Second, abstractions have been applied to allow the verification of properties. I will show how these adjustments are made and why they are allowed.

Chapter 2

Shim6

Multiple efforts have been started to deploy a new multihoming solution that is scalable, can deal with IPv6 addresses and preferably makes no use of PI space. There are two main categories that can be classified: routing-oriented solutions and host-centric solutions. The currently used routing-oriented multihoming approach can also be used in an IPv6 context. As with IPv4, no modifications to the IPv6 architecture are required. This approach generally meets all the goals for multihoming approaches as listed by Abley, et al. [Abl03], except for scalability. Hence, it is no realistic solution. Instead of placing the burden on the global routing tables, host-centric solutions perform the route selection procedure at the end points of the network. They make use of the two semantics of IP addresses. First, an address uniquely identifies an end point in a network. Second, an address acts as a forwarding address (locator) determining the route of an packet. Route selection becomes an outcome of the process of selecting an address for the destination host at the end point, rather than letting routers select the best path, presenting a more scalable solution than routing-oriented solutions.

There are currently three main host-centric efforts [Sav05]: Mobile IPv6, Host Identity Protocol and Site Multihoming by IPv6 Intermediation.

Mobile IPv6

Mobile IPv6 (MIPv6) is designed to support Internet connection for wireless devices such as mobile phones or notebooks. These devices can change the point of attachment to the network. A way to approach the multihoming problem is to compare the situation with such mobile nodes. Nodes that are moving around in the network should remain reachable. Preserving communications through movement could be well compared with preserving communications through outages in multihomed environments. In both cases it must be possible to dynamically changing the paths used while communication maintains unchanged. The MIPv6 protocol [Joh04] already provides this support for mobile nodes, this approach also might be a good solution for multihoming.

MIPv6 separates the end point identifier and locator roles of an IP address. It uses the end point identifier as a stable identifier for the mobile node (referred to as the Home Address (HoA)). This identifier is dynamically mapped to a locator (Care-of Address (CoA)) that corresponds to the current attachment point within the network. A Home Agent is placed in the Home Network to forward packets that are addressed to the HoA, to the current location of the mobile node, as specified by the CoA. After the node moves, it informs the Home Agent about its new location. The mapping between the HoA and the CoA is announced using Binding Update (BU) messages.

The problem here is that the BU message could be forged. The corresponding node may be notified of the new malicious node, and will forward its messages to the attacker. To prevent this, the Return Routeability procedure is designed to allow a correspondent node to authorize the mobile node. By checking if the node is reachable at its HoA and CoA, it can be safely said that the mobile node is really at the foreign link and has a valid registration for the HoA. The lifetime of the binding that is created in the correspondent node using the Return Routeability procedure is limited to seven minutes, in order to make the attack more difficult by periodically checking the validity of the identity.

While the MIPv6 security design could address these redirection threats, unfortunately this procedure cannot be applied to multihoming. MIPv6 relies on HoA's being always reachable, but multihoming design cannot assume this. Consider a link failure at the HoA address. With multihoming, an alternative address is used. The external host is notified of this fact by a BU message. This BU message has to be validated using authorization information obtained through the Return Routeability procedure. The new address pair would only be valid for seven minutes. It is not possible to acquire new authorization information, because communication with the HoA is required, which is no longer reachable.

Host Identity Protocol

The Host Identity Protocol (HIP) enables a strong authentication between hosts at TCP/IP stack level. The protocol is not directly intended for multihoming purposes, but can be used for these kind of scenarios. The HIP architecture introduces a new name space, the Host Identity. HIP would be inserted between the network and transport layers, so that the transport layer can be provided with stable end point identifiers in case of a link failure, and the network layer can transparently change between locators that are linked to the Host Identity.

Unfortunately, introducing a new name space is one major disadvantage of HIP. Applications should use the Host Identity instead of IP addresses. TCP connections and UDP associations are no longer bound to IP addresses but to Host Identities. For multihoming and even communication to work, this means that all other hosts should implement this protocol.

Site Multihoming by IPv6 Intermediation (Shim6)

The IETF started a working group in order to develop a host based multihoming solution. They propose a new sub-layer on the network stack of a host's device. It will enable multihomed hosts to use a set of provider assigned, IPv6 addresses and switch between them transparent for the transport and application layer protocols.

The solution is based on HIP, but the benefit of Shim6 is that changes in the addresses that are used below the shim, are invisible to the upper layer. The fixed address is referred to as upper layer identifier, but does not require a different binding to a new name space. Instead, Shim6 resolves the mapping within the IP protocol.

2.1 Architecture

The mapping of identifiers to locators and backwards can be carried out at different places. In case of a local multihomed network, the process can be performed at the site-exit router. In this case, the host is unaware of the multihomed environment. However, the design intent of Shim6 is to ensure handling path failures independently of the number of IP addresses available to the two communicating hosts, and independently of which host detects the failure condition. This means that the identifier/locator split needs to be performed at the host. This can be done at the transport layer or at the network layer[1]. Multihoming at transport level enables applications to be aware of the currently used address pair. This requires adaptations of applications and transport layer. Besides, addressing is not considered a task for applications. This needs to be done at the network layer. It is suggested that Shim6 operates at this level, within the IP protocol. This is illustrated in Fig. 2.1[2]. Placing the shim at this location, makes the solution transparent and directly usable in a non-multihomed environment without any other adaptations.

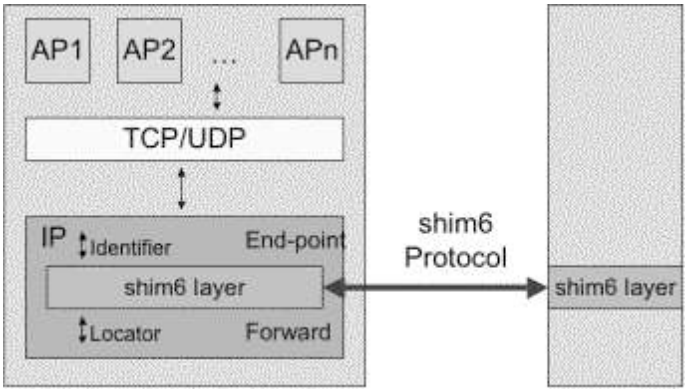

Figure 2.1: Placement of the Shim6 layer in the network stack

Normally, the IP protocol uses the addresses from the transport session as IPv6 addresses. They are considered to be the end points, also called upper layer identifiers (ULID pair). IP constructs a packet consisting of the transport data and the ULID address pair. Additional extension headers may be added. If Shim6 holds information about the default address pair, the host may decide to replace the ULID pair by a different locator pair. Address mapping occurs if there exists multihoming information and the locator pair differs from the ULID pair. This indicates that there

1. Layers refer to the levels of the Open Systems Interconnection Basic Reference Model (OSI Model), an abstract description for computer network protocol design.
2. Picture is taken from the Shim6 website, http://www.shim6.org.

was a problem with the default address pair, and the packet needs to be rewritten. A Shim6 extension header is added to the packet, enabling the receiver to retrieve the correct ULID pair. An IPv6 packet may contain other extension headers. If more than one extension header is used, a specific header order must be used:

- IPv6 header
- Hop-by-Hop Options header
- Destination Options header
- Routing header
- Shim6 header
- Fragment header
- Authentication header
- Encapsulating Security Payload header
- Destination Options header
- Upper Layer header

After the IPv6 header, the routing related headers (Hop-by-Hop Options header and the Routing header) are added. They are placed before the Shim6 data because routers should be able to evaluate those headers, without decoding the Shim6 header. The Fragment header should be placed after the Shim6 header, because when reassembly is carried out at the destination, all source and destination addresses must be identical. The fragments could have been sent over multiple paths, potentially with different source and destination locators. Thus, the locators should be replaced by the ULID pair before defragmenting occurs. Security related headers, such as the Authentication header and the Encrypted Security Payload header, are also below the Shim6 header. This way, IPSec can be made unaware of locator changes, just like the upper layer can be unaware of locator changes. IPSec security associations remain stable, even if the locators are changing.

2.2 Protocol overview

The Shim6 protocol operates in several phases over time. It makes use of the following concepts [Nor06]:

Initial contact Typically, communication starts with an application on host A that wishes to contact an application on host B. For short communications, it may not be worth the overhead of setting up a multihomed environment, since the chance that a failure occurs is very small. Multihoming pays off only for long term communications, and should be negotiated for these types of communications only. Currently, there is no action needed by Shim6.

Context establishment At some point in time, one of the hosts determines that it is useful to set up the multihomed environment. For example, more than 50 packets have already been sent or received. Shim6 initiates a 4-way context establishment. This way, A and B will exchange their list of locators to each other. This information is stored in a local multihomed environment, also called the Shim6 context. If the context establishment fails, the initiator will assume that the other party does not support Shim6. Standard unicast communication can be continued.

Failure detection and locator pair exploration Shim6 maintains information of incoming and outgoing messages. This helps to detect a possible link failure. In case of a failure, one host needs to probe different alternate locator pairs until a new, working address pair is found.

Packet rewriting If a new working locator pair has been found, Shim6 will rewrite the packets on transmit. The packets are tagged with the Shim6 payload extension header, which contains the receiver's context tag. The receiver can use this context tag to find the context state that will indicate which addresses to place in the IPv6 header, before passing the packet to the upper layer protocol (ULP).

Garbage collection When Shim6 thinks that a context is no longer used, it can clean up the state. The context establishment protocol defines a recovery message to signal when there is no context state, so that premature garbage collection or complete state loss (after a crash or reboot) can be recovered.

This algorithm provides redundancy to the hosts, but cannot solve traffic engineering and load sharing. Both features can not yet be realized because the fundamental Shim6 mechanism uses a single current locator pair for each remote host. To solve this, *context forking* is introduced. With context forking, an ULP can specify that a context for let's say the ULID pair (A1, B2) should be forked in two contexts. The two contexts are tagged with a Forked Instance Identifier (FII): the default context has FII zero, the new context will get a non-zero value. An ULP that is aware of the Shim6 context, can share the data over both contexts or choose a preferred context.

2.2.1 Context establishment exchange

The context establishment exchange allows hosts to set up a multihomed environment, or recover from lost contexts. Shim6 contexts are established using a 4-way exchange. This 4-way exchange counters a possible denial of service (DoS) attack. Such an attack can be carried out similarly to a TCP SYN flooding attack [Edd06]. To avoid this attack, the responder will only create a context after the third packet.

The peer's locators might need to be verified during context establishment. The host that owns the ULID must also be the host that uses the relevant locator. Techniques to create hash based addresses (HBA, [Bag05]) or cryptographically generated addresses (CGA, [Aur05]) can be used to do such verification. Another important verification is that the host is indeed reachable at the claimed locator. The latter verification is needed before packets are sent to the locator, the first one must be performed before packets can be received by the peer with the source locator in question.

There are seven states that the Shim6 protocol can reach during context establishment: IDLE (at state machine start), I1-SENT (initiating context establishment exchange), I2-SENT (initiator waiting to complete context establishment exchange), ESTABLISHED (context is established), E-FAILED (context establishment exchange failed), NO-SUPPORT (ICMP[3] Unrecognized Next Header Type is received, indicating that Shim6 is not supported at the receiver) and I2BIS-SENT

3. The Internet Control Message Protocol (ICMP) is used to retrieve information about the health of the network and reports errors about packets that could not be processed properly [Con98].

(potential context loss at the receiver is detected). There are three different types of establishment: Normal context establishment, concurrent context establishment and context recovery. With normal context establishment, one host is the initiator of the exchange and the other acts as the responder. With concurrent context establishment, both hosts want to initiate the exchange. Context recovery is carried out if a context of one host was removed prematurely.

2.2.1.1 Normal context establishment

Normally, the context establishment exchange consists of four messages, as shown in Fig. 2.2. The figure illustrates a communication exchange between host A and host B. In this figure, A will be the initiator of the exchange and B will act as the responder. Every transmitted packet is indicated by a line number. For example, the first packet sent is indicated with (1). If A sends a packet to B, this is denoted with $A \rightarrow B : packet$. The packet consists of different elements, separated by two pipelines ||. At the start of the exchange, both hosts will be in the state IDLE.

The first message $I1$, is the initial message for context establishment. It contains the initiator context tag ($CT(A)$) that A has allocated for the context. Furthermore, it contains a nonce to ensure the message is not replayed. Also, some options can be set to facilitate the protocol. No state will be created yet at the receiver. However, A will set its state to I1-SENT, to track the progress of the exchange. A can retransmit I1 if it does not receive a response on time. After a number of maximum tries, it may assume that the peer does not implement the Shim6 protocol. If it receives back an ICMP error "Unrecognized Next Header", and the included packet is the I1 message, it is a more reliable indication that the other end does not implement Shim6.

(1) $A \rightarrow B : I1 \;||\; CT(A) \;||\; Nonce_1(A) \;||\; options$
(2) $B \rightarrow A : R1 \;||\; Nonce_1(A) \;||\; Nonce_1(B) \;||\; options$
(3) $A \rightarrow B : I2 \;||\; CT(A) \;||\; Nonce_2(A) \;||\; Nonce_1(B) \;||\; options$
(4) $B \rightarrow A : R2 \;||\; CT(B) \;||\; Nonce_2(A) \;||\; options$

Figure 2.2: Normal context establishment

The second message is used in response to the $I1$ message. B replies with the nonce of A and a new nonce to challenge the initiator. The options field contains a responder validator, to verify that the upcoming $I2$ message is indeed sent in response to this $R1$ message and that the parameters in the expected message are the same as those in the $I1$ message. B still does not create a state.

After checking the initiator's nonce, A will send the third message in the context establishment exchange. This message contains again the context tag of A. Furthermore, the response to the nonce of B, a new nonce that should be repeated in message $R2$, the responder validator from the $R1$ message and the locator list of A are included. After sending the message, A sets its state to I2-SENT.

B responds to the $I2$ message and sets its state to ESTABLISHED. The response $R2$ contains the new initiator's nonce, the context tag of B and the locator list of B. When A receives this message, it also updates its state to ESTABLISHED.

The following options are defined:

ULID pair When the IPv6 source and destination addresses in the header do not match the ULID pair, this option must be included. This option contributes to the recovery of a lost context.

Forked Instance Identifier (FII) When another instance of an existing context with the same ULID pair is being created, this option is included to distinguish this new instance from the existing one. The FII is needed for context forking.

Responder Validator To verify that the message is indeed sent in response to a R1 message and that the parameters in the upcoming I2 message are the same as those in the I1 message.

Locator list Optionally set in the $I2$ message when the initiator immediately wishes to tell the responder its list of locators. When sent, required HBA or CGA information for verification must also be included.

Locator preferences Optionally set in the $I2$ message when the locators do not all have equal preferences.

CGA parameter data structure Included when the locator list is set, in order to let the receiver verify the locator list.

CGA signature Included when some of the locators in the list use CGA (and not HBA) for verification.

2.2.1.2 Concurrent context establishment

Normal context establishment is only one way to set up a multihomed environment. It is also possible that both hosts are trying to initiate a context for the same ULID pair. In this case, we might get crossing $I1$ messages. Both hosts will act as initiator and set their state to I1-SENT. Because after message (2), both hosts know they already sent their context tag and a nonce to prevent replay attacks, they can skip the $I2$ and $R1$ messages, and establishing the context by responding with $R2$ messages. The $R2$ message must contain the sender's context tag and the response nonce that was in the initiate message of the other party. Fig. 2.3 shows how this exchange works.

(1) $A \rightarrow B : I1 \mid\mid CT(A) \mid\mid Nonce_1(A) \mid\mid options$
(2) $B \rightarrow A : I1 \mid\mid CT(B) \mid\mid Nonce_1(B) \mid\mid options$
(3) $A \rightarrow B : R2 \mid\mid CT(A) \mid\mid Nonce_1(B) \mid\mid options$
(4) $B \rightarrow A : R2 \mid\mid CT(B) \mid\mid Nonce_1(A) \mid\mid options$

Figure 2.3: Concurrent context establishment: crossing I1 messages

Another possibility is that a responder host has received an $I1$ message and transmitted a $R1$ message. This type of establishment is shown in Fig. 2.4. After message (2), B is waiting for an $I2$ message, but has no state to remember this. In message (3), It is triggered to sent an $I1$ message itself, now acting as an initiator as well. Just after sending, the $I2$ message would finally arrive, resulting in an established context for this host. B sends the final message $R2$ to the other end, so that it also establishes a context. The sixth message in this exchange is triggered because B transmitted an $I1$ message (message (3)), and does not have any effect.

(1) $A \rightarrow B : I1 \mid\mid CT(A) \mid\mid Nonce_1(A) \mid\mid options$
(2) $B \rightarrow A : R1 \mid\mid Nonce_1(A) \mid\mid Nonce_1(B) \mid\mid options$

(3) $B \rightarrow A : I1 \mid\mid CT(B) \mid\mid Nonce_2(B) \mid\mid options$
(4) $A \rightarrow B : I2 \mid\mid CT(A) \mid\mid Nonce_1(B) \mid\mid Nonce_2(A) \mid\mid options$
(5) $B \rightarrow A : R2 \mid\mid CT(B) \mid\mid Nonce_2(A) \mid\mid options$
(6) $A \rightarrow B : R2 \mid\mid CT(A) \mid\mid Nonce_2(B) \mid\mid options$

Figure 2.4: Concurrent context establishment: crossing I1 and I2 messages

2.2.1.3 Context recovery

The exact mechanism to determine when the context state is no longer used is implementation dependent. For example, it might use the existence of ULP state combined with a timer to determine if a state is likely to be no longer used. This means that a situation can occur where one host is still using the context state and the other host does not. If the garbage collection occurred too early, the context should be recovered. There are two possible exchanges that can recover the context. In the first situation, the host that still uses the context state, continues with communication. For example, it probes for alternate locator pairs. This situation is shown in Fig. 2.5.

(1) $A \rightarrow B : payload \mid\mid CT(P)$
(2) $B \rightarrow A : R1bis \mid\mid CT(P) \mid\mid Nonce_1(B) \mid\mid options$
(3) $A \rightarrow B : I2bis \mid\mid CT(A) \mid\mid Nonce_1(A) \mid\mid Nonce_1(B) \mid\mid CT(P) \mid\mid options$
(4) $B \rightarrow A : R2 \mid\mid CT(B) \mid\mid Nonce_1(A) \mid\mid options$

Figure 2.5: Context loss at receiver

The receiver sees a Shim6 message for which it has no context for the received context tag. It notifies the sender by sending a $R1bis$ message. A reduced context establishment is initiated. The $R1bis$ message contains the relevant context tag, and is completed with a nonce and the responder validator option. Host A receives the $R1bis$ message and continues the re-establishment by sending an $I2bis$ message containing the received context tag, its own allocated context tag, the responding nonce, a new nonce and possibly some options again. When receiving the $R1bis$ message, A knows the current context is not longer applicable, so he sets its state from ESTABLISHED to I2BIS-SENT. The context re-establishment is finalized with an $R2$ message transmitted by B, and both ends have an established context again.

It is also possible that the host without context state tries to create a new one (for the same ULID pair), by sending an $I1$ message. The other host finds an existing context, but its allocated context tag does not match. This is referred to as context confusion. In this situation, it leaves the old context established unchanged, but continues a 4-way context establishment with the initiator, like in Fig. 2.2. When receiving the $I2$ message, it removes the old context and accepts the new context associated with the $I2$ message.

2.2.2 Failure detection and locator pair exploration

Failure detection is about detecting reachability of a currently used address pair between two hosts [Ark06]. Reachability is determined by making sure that whenever there is data traffic in one direction, there is also traffic in the other direction. Locator pair exploration is about picking a new address pair to be used when a failure occurs. Failures are relatively infrequent, so the current locator pair that worked a few seconds ago is very likely to be still operational. The mechanism should only invoke heavier exploration when there is a suspected failure. This algorithm is referred to as Forced Bidirectional Detection:

- The algorithm is started after a successful context establishment. The context is considered operational, that is, the currently used locator pair is considered to be working. The algorithm makes use of two timers: (1) a SEND timer to reflect how long ago the last packet was sent, and (2) a KEEPALIVE timer to reflect how long ago the last packet was received. These two timers are mutually exclusive, so at most one timer is running at the time.
- Whenever outgoing data packets are generated that are part of a Shim6 context, the SEND timer is started. If there is a KEEPALIVE timer running, it is stopped.
- Whenever incoming data packets are received, the SEND timer is stopped and the KEEPALIVE timer is started.
- If the KEEPALIVE timer exceeds the keepalive timeout, a keepalive message will be transmitted. A host may send this message sooner (depending on implementation considerations), but the average time after a keepalive message is sent must be at least $keepalivetimeout/2$. Note that if data packets are flowing in both directions, there is no need to keep the connection alive with keepalive messages. No additional keepalive messages are sent by the host, unless a new data packet is received.
- If a keepalive message was received, the timer SEND is stopped. If both ends have sent a keepalive message, the session is idle.
- If the SEND timer exceeds the send timeout, a locator pair exploration is started. The send timeout must be larger than the keepalive timeout to accommodate for lost keepalives and variations in round trip times.

This reachability detection is a form of failure detection. There are two more forms of failure detection that are performed before reachability detection: Tracking local information and tracking remote peer status [Ark06]. Tracking local information consists of using reachability information about the local router as an input. Protocols like Neighbor Discovery, Neighbor Unreachability Detection, Address Autoconfiguration and DHCP can be used to discover and monitor available addresses within the local scope. Tracking remote peer status is important to verify if the peer's currently used address is still in use. If both detection mechanisms succeed, reachability detection is started.

If a failure is detected, an exploration process attempts to find another operational locator pair so that the communication can continue. The party that first detects the problem (because the send timeout expired) starts a process where it sends probe messages to the other party, trying different address pairs each time, until it gets an probe message back. This responding probe message confirms that the initial probe message has arrived and that the other party can reach the host that

detected the problem. The initiator of the exploration process must send another probe message for confirmation to the other host.

(1) $A \rightarrow B : payload \mid\mid (A1, B1)$
(2) $B \rightarrow A : payload \mid\mid (B1, A1)$
Link (A1,B1) failure
(3) $A \rightarrow - : payload \mid\mid (A1, B1)$

(4) $B \rightarrow A : probe \mid\mid Id(p) \mid\mid Exploring \mid\mid (B1, A1)$
(5) $A \rightarrow - : probe \mid\mid Id(q) \mid\mid InboundOK(p) \mid\mid (A1, B1)$
(6) $A \rightarrow B : probe \mid\mid Id(r) \mid\mid InboundOK(p) \mid\mid (A1, B2)$
(7) $B \rightarrow A : probe \mid\mid Id(s) \mid\mid Operational(r) \mid\mid (B1, A1)$
(8) $A \rightarrow B : payload \mid\mid (A1, B2)$
(9) $B \rightarrow A : payload \mid\mid (B1, A1)$

Figure 2.6: Link (A1, B1) failure detected by host B

A failure scenario is given in Fig. 2.6, where host B detects a problem and tries to probe a new locator pair. B sets its SEND timer after sending the payload in message (2). Right after that, the link (A1, B1) that is used by host A fails. B does not see new packets arrive anymore. Note that it will not send a keepalive message, because it transmitted an outgoing data packet. It may send more outgoing packets. When the Send Timeout expires, B sends a complaint that is is not receiving any data packets (message (4)).

In this example, the link (B1, A1) is still working and the probe message will get to A. The message is tagged with an identifier `p` and tells that B does not get data packets from A. Host A realizes that it needs to start the exploration and sends a probe message back. In the example, this message is tagged with the identifier `q` and tells B that A received the probe message `p`. Unfortunately, the link (A1, B1) is not operational at the moment and the message is not delivered. A keeps retransmitting probe messages, until it gets a probe message back. Note that B in the meantime also could have retransmitted his probe messages. In this example, the second probe packet tagged with identifier `r` (message (6)) is received by B. Because the link (B1, A1) still works, B responds on this link with the probe message `p`, this time telling A that he received the probe message `r`. Now A knows that link (A1, B2) is operational and the communication can continue in message (8) and (9) using the links (A1, B2) and (B1, A1).

Detecting failures and the exploration for a new locator pair is called the reachability protocol (REAP). Context establishment and REAP are considered the two most important algorithms for Shim6.

Chapter 3

Formal methods applied to Shim6

In order to verify critical parts of the Shim6 protocol draft specification, it should be described using a formal specification language. I have used UPPAAL [Beh04] to do so. UPPAAL is an integrated tool environment for specification, validation and verification of real time systems modeled as networks of timed automata [Alu94]. Timed automata are finite state machines (FSMs) supplied with clocks. The tool is (1) able to generate a graphical representation of the syntax for FSMs in combination with C-like syntax, (2) it allows you to specify timing constraints and (3) it supports simulation and model checking. These are three important properties that are useful when modeling network protocols.

A system in UPPAAL is composed of one or more processes, each modeled as an automaton. An automaton consists of a set of locations and transitions. Transitions are used to jump between locations and can be utilized with four properties:

a. Using the **select** statement, we can nondeterministically bind a value to an identifier. For example, the statement `x:int[1,m]` binds a value in the interval `[1,m]` to the variable `x`. This means that there is an instance of the transition for each number in this interval.
b. A transition can be utilized with a **guard**. A guard is a condition on variables and clocks, that have to be satisfied before the transition can be taken.
c. The **synchronization** mechanism allows two processes to take a transition at the same time. A synchronization channel `example`, will have an output transition in one process containing `example!`, and an input transition in another process with `example?`. The input transition is taken if and only if the output transition is taken.
d. When taking a transition, some **update** actions are possible. Variables can be assigned a value or clocks can be (re)set. The updates in an output transition will be executed before the assignments in an input transition. This means that variables that are assigned in an output transition can be used in the corresponding input transition.

Fig. 3.1 shows an example UPPAAL automaton. It contains two locations and one transition. The transition is utilized with all possible properties: a select statement `val:int` that nondeterministically selects an integer, a guard `val>4` that ensures

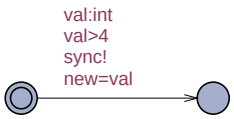

Figure 3.1: An example UPPAAL automaton

the value of `val` is larger than `4`, a synchronization channel `sync!` and an update action `new=val` that assigns the value of `val` to the variable `new`.

Locations can be supplied with invariants. An invariant states an expression that must always be true in that location. For example, if location `P` has an invariant `x<=3` (`x` being a clock), then the process may not stay longer than three time units in `P`. A location may be labeled as *urgent*, which means that it has a designated clock `y`, and contains the invariant `y<=0`. Time may not progress in an urgent state. A *committed* location is even more restrictive. When one of the automata is in a committed location, time may not progress and the next transition has to start from a committed location (from either this or another automaton in the network).

Taking these features into account along with my experience with UPPAAL and the lively research activity for the tool at the Radboud University Nijmegen convinced me that this was the right model checker for modeling network protocols, Shim6 in specific. The most critical parts of Shim6 are the context establishment exchange and the reachability protocol. Context establishment exchange is used to set up a multihomed context with another device and can also be used for context recovery. The reachability protocol is used to detect a failure in a currently used address pair and selecting a new, working address pair.

3.1 Modeling context establishment

The Shim6 draft [Nor06] assumes that Shim6 can provide multihoming between two communicating hosts. A host can have multiple multihoming sessions with different hosts. Our model assumes `h` hosts with `h=2`. This constant variable may be set to a larger value in order to "plugin" additional hosts. A local host maintains at most one Shim6 context per remote host. The draft assumes that a dispatcher delivers incoming packets to context that it belongs to. Host failure, such as a reboot or crash, is not modeled. Host failure means that the host has lost its contexts and is no longer able to send. However, the model allows a host to voluntary stop the transmission of messages. This will eventually lead to the garbage collection of the context. A host that stops sending messages and removes its context can be considered the same behavior as with host failure.

The behavior of each host is modeled by three automata: `Context`, `Dispatcher` and `ULP`. Automaton `Context` models the establishment of a Shim6 context, `Dispatcher` delivers the incoming messages to the corresponding context and the upper layer protocol `ULP` is concerned with sending payload packets. All three automata are parameterized with a host identifier `HostType`. A host can maintain multiple contexts, so the automaton `Context` is parameterized with an additional `HostType` that

identifies the remote host of this context. In reality, the context is identified by a context tag, which can have a value between 0 and 2^{47}. But because *"The context state is maintained per remote ULID i.e. approximately per peer host, and not at any finer granularity"* ([page 10, section 1.6] of the draft), our model identifies the context with just the remote host identifier.

`HostType` is defined as

```
typedef scalar[h] HostType;
```

and denotes the set $\{0, ..., h-1\}$. With a scalar set, the behavior of a model is invariant under arbitrary permutations of the elements of a scalar set [Dil93], [Hen03]. By defining our hosts as a scalar set instead of a subrange, UPPAAL knows that all hosts are fully symmetric, providing the ability to reduce the state space.

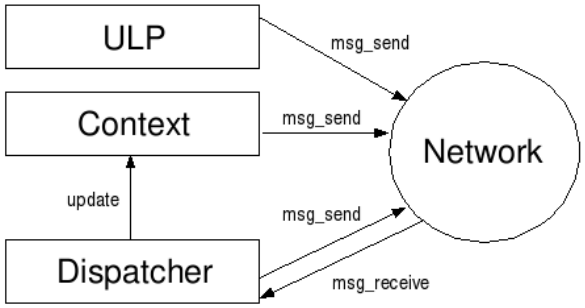

Figure 3.2: Overview of the communication between the host automata and network automaton

The three automata that form a host can communicate with an automaton that represents the network. The network is used to send packets from one host to another. The three automata can also speak with each other mutually. Fig. 3.2 shows how the automata communicate. The automata `ULP`, `Context` and `Dispatcher` can send packets to the network with `msg_send`. Every automata deals with different types of messages. The ULP will only transmit payload packets, the context deals with the sending of initiator Shim6 messages and the dispatcher handles incoming messages and may reply to these with responder Shim6 messages. The `Dispatcher` is the only automata that receives packets through `msg_receive`. It will determine the appropriate action and, if necessary, update the context.

3.1.1 The network

Shim6 makes the following assumptions about the network [page 16, section 3]:

> *The shim6 approach assumes that there are no IPv6-to-IPv6 NATs on the paths, i.e., that the two ends can exchange their own notion of their IPv6 addresses and that those addresses will also make sense to their peer.*

IPv6 [Dee98] is the successor of IPv4. Although IPv6 is not yet widely deployed, all newly deployed applications consider IPv6. The Internet Protocol is used to provide addressing information to data packets. This also applies to Shim6 data packets. For our model, only the source and destination addresses are relevant. The draft

also only considers unicast messages. This allows us to leave out extension headers that provide multicast and anycast addresses.

For the Shim6 environment, an IPv6 packet can be modeled as:

```
typedef struct {
        IPv6Type        src;
        IPv6Type        dest;
        shim6_packet    shim6;
} IPv6_packet;
```

The IPv6 packet data is followed by some Shim6 related data. Every Shim6 message contains a bit P to distinguish payload packets from control messages [page 24, chapter 5]. If the message is a payload packet, the Shim6 header is only supplied with a context tag, so that the receiver can identify the corresponding context. If the message is a control message, the header consists of a type field and some type specific fields. In our model, the P bit is combined with the type field of a control message. If `P=1`, the variable `type` is set to `PAYLOAD`. If `P=0`, the variable `type` is set to a type value as listed in the draft [page 26, section 5.3]. Two additional values, `NO_SHIM` and `ICMP`, are defined. The value `NO_SHIM` specifies there was no Shim6 related data, and the value `ICMP` indicates a special ICMP message (type 4, code 1) that needs to be delivered to the corresponding context. Because UPPAAL does not have C-like unions or pointers, our Shim6 packet is a list of possible data structures, with `type` indicating the structure in use:

```
typedef struct {
        Type type;
        i1_packet i1;
        r1_packet r1;
        i2_packet i2;
        r2_packet r2;
        r1bis_packet r1bis;
        i2bis_packet i2bis;
        updreq_packet updreq;
        updack_packet updack;
        payload_packet payload;
        icmp_packet icmp;
} shim6_packet;
```

The underlying network can be modeled as a set of `n` identical `Network` automata:

```
typedef scalar[n] NetworkType;
```

Each automaton is parametrized by an element from this type. The automata can pass through packets by using synchronization channels. A global packet `p` is defined that is generated in the assignments of an output transition (denoted with `!`), which is then picked up by an input transition (denoted with `?`). Each `Network` automata takes care of handling one packet at a time. This allows us to model race conditions, where messages transmitted later in time can overtake earlier transmitted messages. With `n` automata, `n` messages can be in transit at the same time. In reality, `n` can be a very large number. To reduce state space, we keep the number of `Network` automata as low as possible.

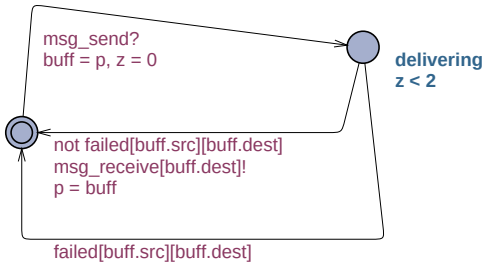

Figure 3.3: Automaton `Network[j]`

The behavior of the network is illustrated in Fig. 3.3. When a host decides to transmit a message, it communicates the data via the `msg_send` synchronization channel. The automaton moves to the `delivering` location. While taking the transition, the IPv6 packet `p` is stored into the local network buffer `buff` and a local clock `z` is reset to zero. Delivery is only possible if the path link is still working: `not failed[buff.src][buff.dest]`. With the synchronization channel `msg_receive[buff.dest]`, the message is delivered to the correct destination. The Shim6 draft does not inform about a possible message delay, but the failure detection draft [Ark06] assumes a maximum one-way delay of 2 seconds. This is realized by adding an invariant `z<2` to the location `delivering`.

3.1.2 The context

The automaton `Context` maintains the progress of context establishment and recovery. A context maintains state variables. The draft proposes a conceptual model of a context structure [page 50, chapter 6]. However, the textual part is inconsistent with the table included in that very same chapter. The table in question introduces new variables that were not mentioned earlier in the text, such as the initiator nonce, the responder nonce and the R1bis context tag. These new variables seem to be necessary for retransmissions during the context establishment, so they are included in our model.

Each context variable is parameterized with two host identifiers (from the local host and the peer), to identify the corresponding `Context` automaton. This is allowed, since every context belongs to one remote host. One disadvantage of this approach is that `h` superfluous contexts are created. Every local host also maintains a context for itself. But if we parameterize the context variables with a context tag, we still would have to deal with the `h` unused contexts. This is because we need one special context tag per host to identify a context that has not yet been allocated.

The `Context` automaton is illustrated in Fig. 3.4. The context establishment exchange starts by sending an I1 message [page 59, section 7.7]:

> *When the shim layer decides to setup a context for a ULID pair, it starts by allocating and initializing the context state for its end. As part of this it assigns a random context tag to the context that is not being used as CT(local) by any other context. Then the initiator can send an I1 message and set the context state to I1-SENT. The I1 message MUST include the ULID pair;*

> *normally in the IPv6 source and destination fields. But if the ULID pair for the context is not used as locator pair for the I1 message, then a ULID option MUST be included in the I1 message.*

Initially, the context is in the location `idle`. A boolean value `heuristics[i][peer]` is used to identify if the shim layer decides to set up a context. If these heuristics become true, the context jumps to the `i1sent` location, and allocates the relevant context variables with the function `allocate_ctx`. The urgent broadcast channel `urg` is used to ensure that the transition is taken as soon as it is enabled. The number of I1 transmissions is set to zero with `tries=0`, and the clock `y` is set to `I1_TIMEOUT`, so that the first I1 message is sent immediately. This is realized by the invariant `y<=I1_TIMEOUT` in location `i1sent`. The message is created with the function `send_i1` and is passed on directly to the network.

If the host receives no valid response, it needs to retransmit I1 [page 60, section 7.8]:

> *If the host does not receive an I2 or R2 message in response to the I1 message after I1_TIMEOUT time, then it needs to retransmit the I1 message.*
>
> *If, after I1_RETRIES_MAX retransmissions, there is no response, then most likely the peer does not implement the shim6 protocol, or there could be a firewall that blocks the protocol. In this case it makes sense for the host to remember to not try again to establish a context with that ULID. However, any such negative caching should retained for at most NO_R1_HOLDDOWN_TIME, to be able to later setup a context should the problem have been that the host was not reachable at all when the shim tried to establish the context.*
>
> *If the host receives an ICMP error with "Unrecognized Next Header" type (type 4, code 1) and the included packet is the I1 message it just sent, then this is a more reliable indication that the peer ULID does not implement shim6. Again, in this case, the host should remember to not try again to establish a context with that ULID. Such negative caching should retained for at most ICMP_HOLDDOWN_TIME, which should be significantly longer than the previous case.*

The invariant `y<=I1_TIMEOUT` also forces the context to retransmit I1. The counter `tries` is incremented to maintain the number of retransmissions. If `tries > I1_RETRIES_MAX` and within `I1_TIMEOUT` still no valid response is received, the context jumps to the location `failed`. The clock `y` is reset to zero and variable `to` is set to `NO_R1_HOLDDOWN_TIME`. If the context is informed that an ICMP error (type 4, code 1) is received, the context also jumps to `failed`, but now `to` is set to `ICMP_HOLDDOWN_TIME`. After the timeout occurs, the context can be removed, that is, it can be set to `idle` again. If a R1 message was received on time, the host will continue the exchange by transmitting I2 [page 62, section 7.11]:

> *Upon the reception of an R1 message, the host extracts the Initiator Nonce and the Locator Pair from the message (the latter from the source and destination fields in the IPv6 header). Next the host looks for an existing context which matches the Initiator Nonce and where the locators are contained in Ls(peer) and Ls(local), respectively. If no such context is found, then the R1 message is silently discarded. If such a context is found, then the host looks at the state: If the state is I1-SENT, then it sends an I2 message as specified below.*
>
> *When the host sends an I2 message, then it includes the Responder Validator option that was in the R1 message. The I2 message MUST include the ULID pair; normally in the IPv6 source and destination fields. If a ULID-pair*

> *option was included in the I1 message then it MUST be included in the I2 message as well. Besides, the I2 message contains an Initiator Nonce. This is not required to be the same than the one included in the previous I1 message.*

The context is notified of the receipt of R1 through synchronization channel `update[i][peer][I2SENT]?`. Note that all outgoing transitions are guarded by `tries>0`. This forces the host to send at least one I1 packet, before updating its context. The clock is now set to `I2_TIMEOUT`, so that the I2 message is sent immediately. The message is constructed in `send_i2`. All information is retrieved from the context variables, except the responder nonce and responder validator, which are copied from the triggering message. In addition, the host includes its locator list.

I2 messages may also be retransmitted [page 63, section 7.12]:

> *If the initiator does not receive an R2 message after I2_TIMEOUT time after sending an I2 message it MAY retransmit the I2 message, using binary exponential backoff and randomized timers. The Responder Validator option might have a limited lifetime, that is, the peer might reject Responder Validator options that are older than VALIDATOR_MIN_LIFETIME to avoid replay attacks. Thus the initiator SHOULD fall back to retransmitting the I1 message when there is no R2 received after retransmitting the I2 message I2_RETRIES_MAX times.*

Retransmissions of I2 messages occur similar to retransmissions of I1 messages. Different is that if after `I2_RETRIES_MAX` no valid response is received, the context should fall back to `i1sent`, instead of moving to `failed`. An implementation may decide not to retransmit I2 messages. This is simulated by the transition that does nothing else than reset the clock `y`. This way, the context state remains `i2sent`, without the invariant of the location becoming unsatisfied.

The `VALIDATOR_MIN_LIFETIME` value is not modelled, since the behavior is not well-explained in the draft. For instance, it is not clear how the responder can maintain such a lifetime.

In the case of concurrent establishment, a context in location `i1sent` may receive an I2 or I2bis message. In this case, the context jumps to `established`. In location `i2sent`, the context can move to established on receipt of an R2 message [page 66, section 7.16]:

> *If state is I1-SENT, I2-SENT, or I2BIS-SENT then the host performs the following actions: If a CGA Parameter Data Structure (PDS) is included in the message, then the host MUST verify that the actual PDS contained in the message corresponds to the ULID(peer) as specified in Section 7.2. If the verification fails, then the message is silently dropped. If the verification succeeds, then the host records the information from the R2 message in the context state; it records the peer's locator set and CT(peer). The host SHOULD perform the HBA/CGA verification of the peer's locator set at this point in time, as specified in Section 7.2. The host sets its state to ESTABLISHED.*

The model abstracts from the CGA verification, so from `i1sent`, `i2sent` and `i2bissent`, the context can immediately jump to `established`. The function `establish_ctx` records the information from the R2 message. The context clock is reset to zero. From now on, the clock `y` determines when to garbage collect the context. When this clock reaches `TEARDOWN_TIMEOUT`, the context is triggered to be

removed. According to the Shim6 draft, `TEARDOWN_TIMEOUT` should be set to 300 seconds [page 73, section 9]:

> *Thus it is RECOMMENDED that implementations minimize premature teardown by observing the amount of traffic that is sent and received using the context, and only after it appears quiescent, tear down the state. A reasonable approach would be not to tear down a context until at least 5 minutes have passed since the last message was sent or received using the context.*

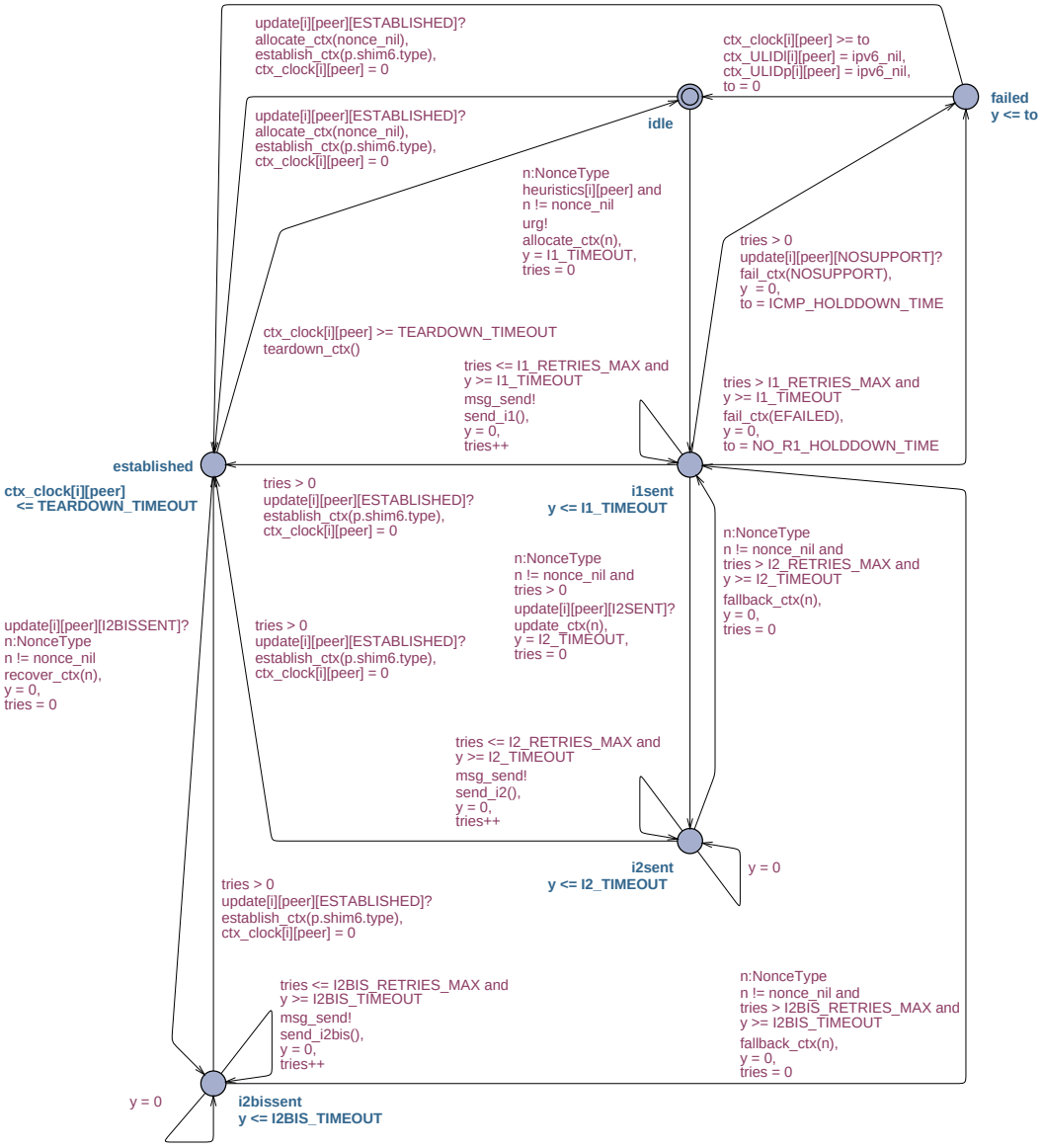

Figure 3.4: Automaton `Context[i][peer]`

If a host has removed its context prematurely and it receives a Shim6 payload, its context must be recovered. The host notifies the peer by sending a R1bis message. When receiving such a message, the host detects that the peer context was lost and sets its context state to I2BIS-SENT . The behavior in the location `i2bissent` is almost identical to the behavior in `i2sent`. The difference is that the upper layer with a context in `i2bissent` may still send Shim6 payload messages. On receipt of a R2 message, the context can jump back to `established`.

Until now, we have seen the the behavior of a host that acts as initiator. If a host acts as responder, it stays in `idle` until the receipt of an I2 or I2bis message. The synchronization channel `update[i][peer][ESTABLISHED]?` is used to inform the corresponding context. Upon this synchronization, the context jumps to the `established` location. The context should behave identical when it is in the location `failed` instead of `idle`.

3.1.3 The dispatcher

Determining which actions to fulfill when receiving a Shim6 control message is elaborately discussed in chapter 7 of the draft. Basically, every message could result in three possible actions: (1) updating a context, (2) resolving context confusion, and (3) generating a reply message. But first, the host needs to look if there is a context that corresponds to the incoming message [page 81, section 12.3]:

> *We assume that each shim context has its own state machine. We assume that a dispatcher delivers incoming packets to the state machine that it belongs to. Here we describe the rules used for the dispatcher to deliver packets to the correct shim context state machine. There is one state machine per context identified that is conceptually identified by ULID pair and Forked Instance Identifier (which is zero by default), or identified by CT(local). However, the detailed lookup rules are more complex, especially during context establishment. Clearly, if the required context is not established, it will be in IDLE state. During context establishment, the context is identified as follows:*
>
> *- I1 packets: Deliver to the context associated with the ULID pair and the Forked Instance Identifier.*
>
> *- I2 packets: Deliver to the context associated with the ULID pair and the Forked Instance Identifier.*
>
> *- R1 packets: Deliver to the context with the locator pair included in the packet and the Initiator nonce included in the packet (R1 does not contain ULID pair nor the CT(local)). If no context exist with this locator pair and Initiator nonce, then silently discard.*
>
> *- R2 packets: Deliver to the context with the locator pair included in the packet and the Initiator nonce included in the packet (R2 does not contain ULID pair nor the CT(local)). If no context exists with this locator pair and INIT nonce, then silently discard.*
>
> *- R1bis packet: deliver to the context that has the locator pair and the CT(peer) equal to the Packet Context Tag included in the R1bis packet.*
>
> *- I2bis packets: Deliver to the context associated with the ULID pair and the Forked Instance Identifier.*
>
> *- Payload extension headers: Deliver to the context with CT(local) equal to the Receiver Context Tag included in the packet.*
>
> *- ICMP errors which contain a Shim6 payload extension header or other shim control packet in the "packet in error": Use the "packet in error" for dis-*

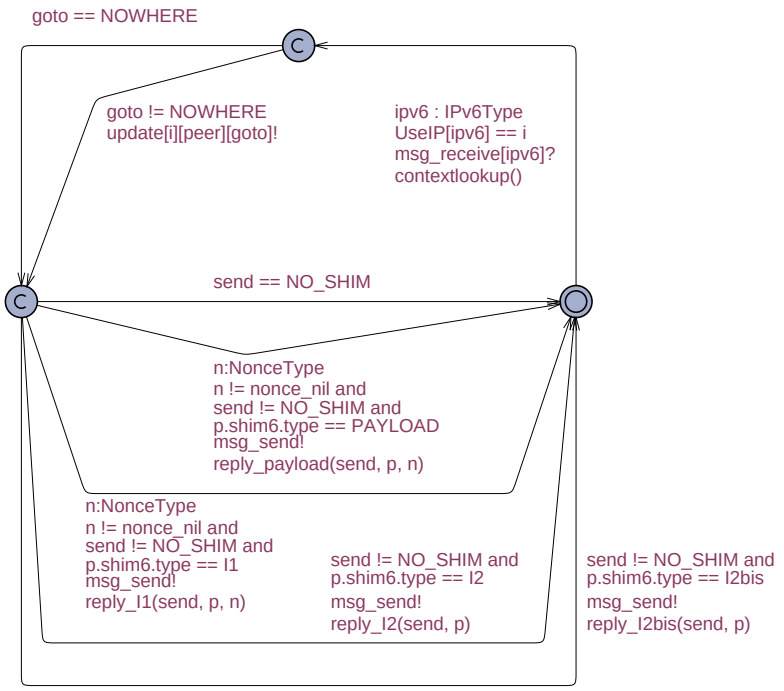

Figure 3.5: Automaton `Dispatcher[i]`

> *patching as follows. Deliver to the context with CT(peer) equal to the Receiver Context Tag, Lp(local) being the IPv6 source address, and Lp(peer) being the IPv6 destination address.*

These rules are implemented in the function `contextlookup`. Note that in our model the context is identified by the peer host identifier. We do not need to *find* the correct context, but we do need to *verify* that the relevant context variables are set.

For example, an incoming I1 message is checked on its ULID pair and its FII:

```
if (p.shim6.type == I1) {
  if (p.src == ctx_ULIDp[i][peer] and
      p.dest == ctx_ULIDl[i][peer] and
      p.shim6.i1.FII == ctx_FII[i][peer]) {

    ...
  }
  else if (ctx_state[i][peer] == IDLE or
           ctx_state[i][peer] == EFAILED or
           ctx_state[i][peer] == NOSUPPORT) {
   ...
  }
}
```

The body of the second `if` and `else` statement (indicated by the dots) then fulfills

the required actions. On receipt of a message, the dispatcher performs the context lookup, as illustrated in Fig. 3.5. The function `contextlookup` sets two variables: `goto` and `send`. The variable `goto` determines if the context state needs to be updated, while `send` determines which message needs to be replied. When receiving an I1 message, the host usually responds with R1 [page 60, section 7.9]:

> *If no state is found (i.e., the state is IDLE), then the host replies with a R1 message as specified below.*
>
> *If such a context exists in ESTABLISHED state, the host verifies that the locator of the Initiator is included in Ls(peer). If the state exists in ESTABLISHED state and the locators do not fall in the locator sets, then the host replies with a R1 message as specified below.*
>
> *If the state exists in ESTABLISHED state and the locators do fall in the sets, then the host compares CT(peer) for the context with the CT contained in the I1 message.*
>
> *- If the context tags match, then this probably means that the R2 message was lost and this I1 is a retransmission. In this case, the host replies with a R2 message containing the information available for the existent context.*
>
> *- If the context tags do not match, then it probably means that the Initiator has lost the context information for this context and it is trying to establish a new one for the same ULID-pair. In this case, the host replies with a R1 message as specified below.*
>
> *If the state exists in other state (I1-SENT, I2-SENT, I2BIS-SENT), we are in the situation of Concurrent context establishment described in Section 7.4. In this case, the host leaves CT(peer) unchanged, and replies with a R2 message. This completes the I1 processing, with the context state being unchanged.*

The body of the `if` statement can be implemented according to this described behavior:

```
if (ctx_state[i][peer] == ESTABLISHED) {
  if (ctx_Lsp[i][peer][p.src] and
      p.shim6.i1.ct_init == ctx_CTp[i][peer])

    send = R2;
  else
    send = R1;
}
else if (ctx_state[i][peer] == I1SENT or
         ctx_state[i][peer] == I2SENT or
         ctx_state[i][peer] == I2BISSENT)

  send = R2;
else if (ctx_state[i][peer] == EFAILED
         or ctx_state[i][peer] == NOSUPPORT)
  send = R1;
```

In this example, only the variable `send` is set. This is because the receipt of I1 messages will never result in an update of the context. Depending on the context state, the dispatcher will reply a R1 or R2 message. For every type of control message, the draft describes the corresponding actions. Each type of control message can be converted to UPPAAL syntax similarly as we did for I1. Also, payload packets might lead to a response [page 80, section 12.1]:

> *The receiver extracts the context tag from the payload extension header, and uses this to find a ULID-pair context. If no context is found, the receiver SHOULD generate a R1bis message (see Section 7.17). Then, depending on the state of the context:*
> *- If ESTABLISHED: Proceed to process message.*
> *- If I1-SENT, discard the message and stay in I1-SENT.*
> *- If I2-SENT, then send R2 and proceed to process the message.*
> *- If I2BIS-SENT, then send R2 and proceed to process the message.*

This logic results in the following UPPAAL code:

```
if (p.shim6.type == PAYLOAD) {
  if (p.shim6.payload.ct_recv != ct_nil and
      p.shim6.payload.ct_recv == ctx_CTl[i][peer]) {

    if (ctx_state[i][peer] == ESTABLISHED)
      ctx_clock[i][peer] = 0;
    else if (ctx_state[i][peer] == I2SENT or
             ctx_state[i][peer] == I2BISSENT) {

      ctx_clock[i][peer] = 0;
      send = R2;
    }
    else if (ctx_state[i][peer] == EFAILED or
             ctx_state[i][peer] == NOSUPPORT)

      send = R1bis;
    }
    else if (ctx_state[i][peer] == IDLE or
             ctx_state[i][peer] == EFAILED or
             ctx_state[i][peer] == NOSUPPORT)
      send = R1bis;
    }
}
```

Finally, the host must check if no context confusion occurred [page 65, section 7.15]. However, context confusion occurs if there are two or more contexts for the same remote host. Since our model maintains exactly one context for each remote host, we have already abstracted from context confusion.

3.1.4 The upper layer protocol

The layer above Shim6 is concerned with sending payloads. Fig. 3.6 displays the automaton `ULP[i]`, which specifies the sending of payload messages.

At this moment, the host is able to send payload packets. One goal of Shim6 is to *"Not require extra roundtrip up front to setup shim specific state. Instead allow the upper layer traffic (e.g., TCP) to flow as normal and defer the setup of the shim state until some number of packets have been exchanged."* [page 5, section 1.1]. We also assume that the upper layer is not aware of the Shim6 layer, so context forking is out of scope.

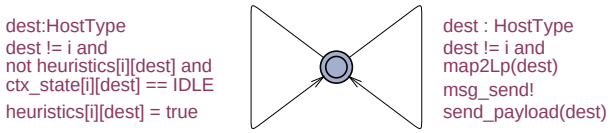

Figure 3.6: Automaton `ULP[i]`

Sending of shim unaware payloads is not modeled. It does not contribute to the context establishment exchange. Instead, at a certain point of time the host decides to set the heuristic trigger by taking the left transition. It is proposed in the draft that the Shim6 context is set up after receiving or sending 50 payload packets. Our approach skips this proposal to avoid long simulation traces.

If the context establishment succeeds and there is need for locator mapping, the host can transmit Shim6 enabled payloads by taking the right transition [page 78, section 11]:

> *When there is no context state for the ULID pair on the sender, there is no effect on how ULP packets are sent. If the host is using some heuristic for determining when to perform a deferred context establishment, then the host might need to do some accounting (count the number of packets sent and received) even before there is a ULID-pair context.*
>
> *If the context is not in ESTABLISHED or I2BIS-SENT state, then it there is also no effect on how the ULP packets are sent. Only in the ESTABLISHED and I2BIS-SENT states does the host have CT(peer) and Ls(peer) set.*
>
> *If there is a ULID-pair context for the ULID pair, then the sender needs to verify whether context uses the ULIDs as locators, that is, whether Lp(peer) == ULID(peer) and Lp(local) == ULID(local).*
>
> *If this is the case, then packets can be sent unmodified by the shim. If it is not the case, then the logic in Section 11.1 will need to be used.*

The right transition of the `ULP` automaton is guarded by the function `map2Lp`. This ensures that only Shim6 enabled payloads are being transmitted:

```
bool map2Lp(HostType peer) {
        if (ctx_state[i][peer] == ESTABLISHED or
            ctx_state[i][peer] == I2BISSENT) {

          return true;
        }
        return false;
}
```

Checking if the locator pair equals the ULID pair is not necessary, because update messages and probe messages are not transmitted during context establishment, so the locator pair will always be the ULID pair. This means that in this model only Shim6 aware payload packets without a Shim6 header will be transmitted.

Section 11.1 continues:

> *When sending packets, if there is a ULID-pair context for the ULID pair, and the ULID pair is no longer used as the locator pair, then the sender needs to transform the packet. Apart from replacing the IPv6 source and destination*

> *fields with a locator pair, an 8-octet header is added so that the receiver can find the context and inverse the transformation.*
>
> *If there has been a failure causing a switch, and later the context switches back to sending things using the ULID pair as the locator pair, then there is no longer a need to do any packet transformation by the sender, hence there is no need to include the 8-octet extension header.*
>
> *First, the IP address fields are replaced. The IPv6 source address field is set to Lp(local) and the destination address field is set to Lp(peer). The inserted shim6 Payload extension header includes the peer's context tag.*

If there exists a context for the ULID pair, and the context is in state `established` or `i2bissent`, the address fields may need to be replaced. When the ULID pair is equal to the currently used locator pair, the packet does not need to be modified. Otherwise, the logic of section 11.1 of the Shim6 document should be executed. This can be translated into five assignments: the source field is assigned with the local locator, the destination field is assigned with the peer locator, the type is set to `PAYLOAD` to indicate a payload extension header, the receiver context tag is set to the peer context tag, and the context clock is reset to zero, in order to let the context know it is still being used. The function `pzero` clears the IPv6 packet `p`, to make sure that all the irrelevant fields are empty:

```
void send_payload(HostType dest) {
        pzero();

        p.src = ctx_ULIDl[i][dest];
        p.dest = ctx_ULIDp[i][dest];
        p.shim6.type = PAYLOAD;
        ctx_clock[i][dest] = 0;

        if (ctx_ULIDl[i][peer] != ctx_Lpl[i][peer] or
            ctx_ULIDp[i][peer] != ctx_Lpp[i][peer]) {

          p.src = ctx_Lpl[i][dest];
          p.dest = ctx_Lpp[i][dest];
          p.shim6.payload.ct_recv = ctx_CTp[i][dest];
        }
}
```

Receiving payloads at the upper layer is not modeled, since passing on payload packets to the upper layer does not affect the behavior of the shim6 protocol.

3.1.5 Initializing the model

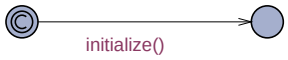

Figure 3.7: Automaton `Initializer`

Before hosts are able to communicate, they need to be assigned IPv6 addresses:

```
typedef scalar[m] IPv6Type;
```

Just like `HostType`,IPv6 addresses are also scalar sets. Scalar sets provide symmetry reduction, but only permit restricted operations. With scalar sets, only assignment and identity testing are allowed. But because both types are unordered, we cannot simply assign IPv6 addresses to hosts. To provide every host of one or more IPv6 addresses, a new automaton `Initializer` is introduced. As illustrated in Fig. 3.7, it consists of only one transition and two locations. Because the initial location is committed, this transition will always be taken first, and exactly once. The transition calls the function `initialize`. In this function, each IPv6 address is assigned to a list element. The same algorithm is used for host identifiers. This results in two ordered lists, `IP` and `Host`. We now have created ordered scalar sets, and are now able to assign addresses to our hosts:

```
for(j=0;j<m;j++) {
        if (count >= h) {
                ulid_done = true;
                count = 0;
        }
        UseIP[IP[j]] = Host[count];

        if (!ulid_done)
                ULID[Host[count]] = IP[j];

        count++;
}
```

First the variable `count` is verified to be smaller than the number of hosts, otherwise an IPv6 address would be assigned to an undefined host. If `count` is greater, it is reset. With `UseIP[IP[j]] = Host[count]`, the actual assignment is fulfilled. Finally, the first `h` addresses are defined as the upper layer identifiers of the hosts: `ULID[Host[count]]=IP[j]`.

The values for the timing constants [page 84, chapter 14] can be copied verbatim in the UPPAAL declarations section:

```
const int I1_RETRIES_MAX = 4;
const int I1_TIMEOUT = 4; // seconds
const int NO_R1_HOLDDOWN_TIME = 60; // seconds
const int ICMP_HOLDDOWN_TIME = 600; // seconds
const int I2_TIMEOUT = 4; // seconds
const int I2_RETRIES_MAX = 2;
const int I2BIS_TIMEOUT = 4; // seconds
const int I2BIS_RETRIES_MAX = 2;
const int VALIDATOR_MIN_LIFETIME = 30; // seconds
const int UPDATE_TIMEOUT = 4; // seconds
```

3.2 Adding REAP

When the context establishment exchange is executed successfully, REAP is started. This is vital functionality that provides redundancy to your network connection and

is therefore also a critical part of the Shim6 protocol. For this reason, formal methods should also be applied to REAP. Preferably, it is integrated within the context establishment model. Unfortunately, verification (described in the next chapter) already revealed state space problems with the current model. That's why I decided to verify the two algorithms as two separate models. This is perfectly acceptable, because both algorithms can perform independently.

Some automata of the context establishment model can be used for REAP as well. The automaton `Initializer` that provides hosts with IP addresses, is also important to REAP. Also, the `Network` automaton can be re-used for packet transmission. The modeled network already has methods to check if the used path encountered a problem, but the system does not yet provide ways to create a link failure. This was not necessary for context establishment, but it is an important variable for failure detection and new locator pair exploration. A new automaton `Fail` is introduced, illustrated in Fig. 3.8. This automaton allows the system to fail a path that is valid with respect to the address pairs. Valid address pairs do not contain the special IPv6 address `ipv6_nil` and assure that the two addresses are used by different hosts: `UseIP[in] != UseIP[out]`. A link failure is created with `failed[in][out]=true`. A simplification in this system is that links cannot fail more than once. In reality, links can go down and recover quickly. A host might not even notice that there was such a small problem. That is just why this simplification is made. We would like to have the ability to verify that a host detects an occurring link failure. By adding link recovery, it gets more complicated to identify the cases when a link failure should or should not be detected.

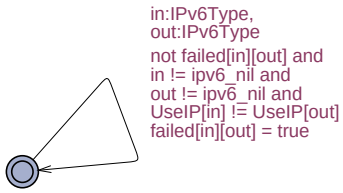

Figure 3.8: Automaton `Fail`

3.2.1 Failure detection and locator pair exploration

An established Shim6 context maintains reachability state and information and can be implemented as a state machine. A host maintains exactly one reachability state machine per Shim6 context. This is modeled by the automaton `REAP`. We continue the use of a dispatcher that informs the state machine of incoming messages and may send response messages. the `REAP` and `Dispatcher` automata form the behavior of the host and are parameterized with the host identifier `HostType`.

The automaton `Dispatcher` is much simpler than the dispatcher of the context establishment. Basically, it receives a message from the network, passes the message on to the `REAP` automaton, and sends a message back if `REAP` generated a reply. In case a reply message is needed, a global boolean `sending` is set to `true`, that is read out by the dispatcher. The new dispatcher is shown in Fig. 3.9.

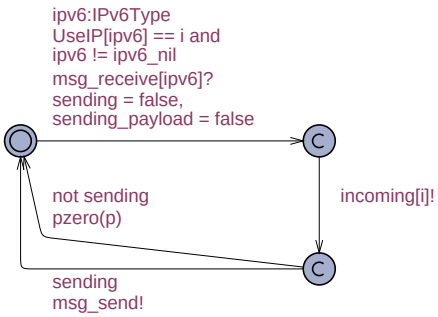

Figure 3.9: Automaton `Dispatcher[i]`

The automaton `REAP` is illustrated in Fig. 3.10. It contains four locations: `Operational`, `Exploring`, `InboundOK` and the initial location. In the location `Operational`, the current address pair is assumed to be working. In the location `Exploring`, the host has currently not seen any traffic and suspects a link failure. In the location `InboundOK`, the host is aware of the problem that the peer encountered, but itself does see incoming traffic. The initial location set ups the REAP state and information. It could be conceived as a very fast, abstracted form of context establishment, that enables reachability detection. The transition initializes the current address pair, sets the timer inactive and moves to the location `Operational`.

Two timers are maintained: the send timer and the keepalive timer. The send timer, reflects to the time that the last payload packet was sent. The keepalive timer reflects to the time the last payload packet was received. The timers are mutually exclusive, that is, they cannot run both at the same time. These timers are modeled by a clock `x`. The automata uses another clock `y` for retransmissions. To ensure that certain actions are taken at the correct time, the locations are supplied with an invariant. For example in the location `Operational`, the send timer may not exceed the send timeout, the keepalive timer may not exceed the keepalive timeout and keepalive messages are retransmitted within a certain interval:

```
(timer == T_SEND imply x <= SEND_TIMEOUT) and
(timer == T_KEEPALIVE imply (y <= KEEPALIVE_INTERVAL and
                             x <= KEEPALIVE_TIMEOUT))
```

REAP becomes truly active if the send or keepalive timer is running. This happens if payload is exchanged between the two communicating hosts [page 23, section 6.2]:

> *Upon sending a payload packet in the Operational state, the node stops the Keepalive timer if it was running and starts the Send timer if it was not running. In the Exploring state there is no effect, and in the InboundOK state the node simply starts the Send timer if it was not yet running.*

The location `Operational` is provided with a transition to itself that takes care of sending payload. The payload message is constructed with `send(PAYLOAD)` and is transmitted through the synchronization channel `msg_send`. The send timer is started with `start_timer(T_SEND)`. It is not necessary to stop the keepalive timer here. Since the timers are mutually exclusive, the keepalive timer is stopped automatically if the send timer is started. In our model, payload is only sent to activate reachability detection.

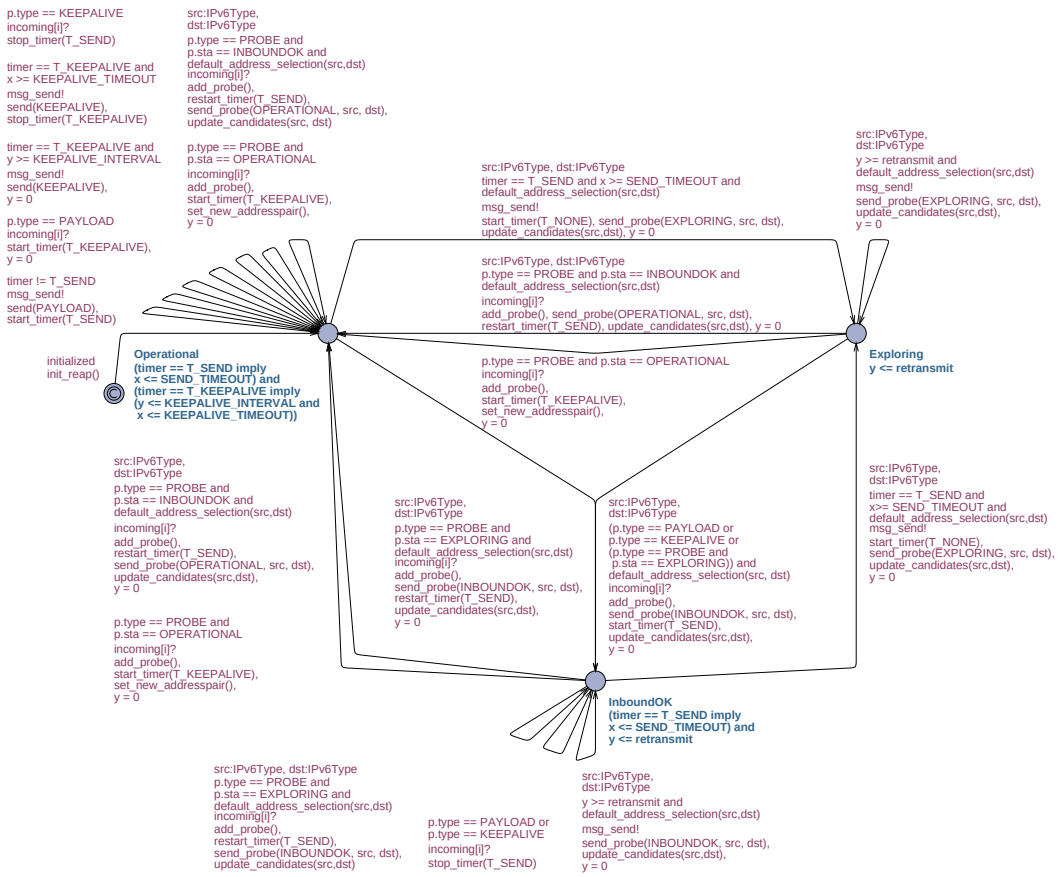

Figure 3.10: Automaton `REAP[i]`

This way, we prevent the payload taking the upper hand in the protocol. Our goal here is to focus on the REAP messages and their ability to detect failures and agree on new locator pairs. That is why the transitions is guarded by `timer!=T_SEND` and sending payload is not possible in the locations `InboundOK` and `Exploring`.

Incoming payload may stop the send timer again [page 22, section 6.1]:

> *Upon the reception of a payload packet in the Operational state, the node starts the Keepalive timer if it is not yet running, and stops the Send timer if it was running.*
>
> *If the host is in the Exploring state it transitions to the InboundOK state, sends a Probe message, and starts the Send timer. It fills the Psent and corresponding Probe source address, Probe destination address, Probe nonce, and Probe data fields with information about recent Probe messages that have not yet been reported as seen by the peer. It also fills the Precvd and corresponding Probe source address, Probe destination address, Probe nonce, and Probe data fields with information about recent Probe messages it has seen from the peer. When sending a Probe message, the State field MUST be set to a value that matches the conceptual state of the sender after sending the Probe. In this case the node therefore sets the Sta field to 2 (InboundOk).*
>
> *In the InboundOK state the node stops the Send timer if it was running,*

but does not do anything else.

The transition from `Exploring` to `InboundOK` is the most interesting here. The exploring host eventually receives incoming messages, so it may move away from the `Exploring` location. Incoming messages are notified by the synchronization channel `incoming[i]`, where `i` denotes the host identifier. The incoming message may be a payload, but also keepalive messages and exploring probes trigger the host to move to `InboundOK`. If the incoming message is a probe, the received probe information is stored with the function `add_probe`. The host will notify the peer that it sees incoming traffic by sending an "Inbound OK" probe. According to the draft, the IPv6 addresses should be selected according to the default address selection algorithm [Dra03]. This algorithm checks all kinds of address properties, such as their scope and label, whether they are deprecated, wheter they serve as home address, and so on. Since such verifications cause many additional variables and states in our model, our selection procedure is simplified. The host randomly selects a `src` and `dst` IPv6 address that are a candidate according to the function `default_address_selection`:

```
bool default_address_selection(IPv6Type src, IPv6Type dst) {
        if (src == ipv6_nil or dst == ipv6_nil)
          return false;
        if (UseIP[src] != i or UseIP[dst] == i)
          return false;

        return candidate[src][dst];
}
```

This function first ensures that the address pair is valid. This means that the source or destination may not be the special IPv6 address `ipv6_nil`, the source address is in use by host `i` and the destination address is not in use by host `i`. Finally, the function looks up the value `candidate[src][dst]`, that stores whether address pair `(src,dst)` is a candidate address pair. The function `update_candidates` makes sure that the used address pair is not a candidate anymore. This way, the same address pair is not selected the next time and all possible address pairs will be used during the exploration phase. If there are no candidates left, this function also takes care of resetting the candidates. When transitioning to `InboundOK`, the host starts the send timer and resets clock `y`, that is needed for retransmissions.

In the locations `Operational` and `InboundOK`, the actions for receiving payload are much easier. These transitions are guarded by `p.type==PAYLOAD`, to ensure the incoming packet is a payload. In case the model is in `InboundOK`, solely the send timer is stopped (`stop_timer(T_SEND)`). In case the model is operational, the keepalive timer is started (`start_timer(T_KEEPALIVE)`) and a clock `y` is set to zero.

Clock `y` is needed to send keepalive messages on a certain interval [page 23, section 6.1]:

> *While the Keepalive timer is running, the node SHOULD send Keepalive messages to the peer with an interval of Keepalive Interval seconds. Conceptually, a separate timer is used to distinguish between the interval between Keepalive messages and the overall Keepalive Timeout interval.*

This is realized by the transition guarded with `timer==T_KEEPALIVE` and `y>=KEEPALIVE_INTERVAL`. The host sends a keepalive with `send(KEEPALIVE)` and resets the clock `y`. When the keepalive timeout is reached, the host sends one last keepalive and stops the keepalive timer. When the send timeout is reached, the host suspects a problem [page 24, section 6.4]:

> *Upon a timeout on the Send timer, the node enters the Exploring state, sends a Probe message, and stops the Keepalive timer if it was running.*

If a problem is detected, the host moves to the location `Exploring`. These transitions may only be taken if `timer==T_SEND` and `x>=SEND_TIMEOUT`. The host will send an exploring probe to the peer. The IPv6 addresses are again selected by our simplified default address selection procedure. Since no timers run in the Exploring state, the timer is made inactive with `start_timer(T_NONE)`. Clock `y` is reset with possible retransmissions in mind.

If the host received a keepalive within `SEND_TIMEOUT` time, it stops the send timer. From this point, the state machine is considered idle until new payload is transmitted or received. Otherwise, a probe message exchange is started. On receipt of an exploring probe, the host moves to the `InboundOK` location [page 25, section 6.7]:

> *Upon receiving a Probe with State set to Exploring, the node enters the InboundOK state, sends a Probe as described in Section 6.1, stops the Keepalive timer if it was running, and restarts the Send timer.*

If the host receives a probe message indicating that the peer is in the `InboundOK` location, it may assume that a new working locator pair is found [page 25, section 6.8]:

> *Upon the reception of a Probe message with State set to InboundOk, the node sends a Probe message, restarts the Send timer, stops the Keepalive timer if it was running, and transitions to the Operational state.*

The transmitted probe message has its state set to Operational, and is meant to inform the peer of the new working locator. On receipt of such probe, the exploration process is terminated [page 25, section 6.9]:

> *Upon the reception of a Probe message with State set to Operational, the node stops the Send timer if it was running, starts the Keepalive timer if it was not yet running, and transitions to the Operational state.*

This final message contains the new working locator in the probe received field. The address pair is updated with the function `set_new_addresspair`:

```
void set_new_addresspair() {
        curr_local = p.received.src;
        curr_peer = p.received.dest;

        last_sent.src = ipv6_nil;
        last_sent.dest = ipv6_nil;
        last_recvd.src = ipv6_nil;
        last_recvd.dest = ipv6_nil;

        /* reset all candidates */
        if (forall(j:IPv6Type)forall(k:IPv6Type)(not candidate[j][k])) {
          for(j:IPv6Type)
```

```
            for(k:IPv6Type)
              if (UseIP[j] == i and UseIP[k] != i and
                  j != ipv6_nil and k != ipv6_nil)

                candidate[j][k] = true;
        }
}
```

The function also clears the reports about last send and received probe messages and resets all candidate address pairs. Our models are now ready for verification. The models are also shown in App. B.

Chapter 4

Verification

The Shim6 draft is unclear about what properties the protocol must satisfy. We may assume that if the amount of packet loss is low, two hosts must be able to establish a context after one of them has been triggered to setup a context state. Also, the protocol must contain no deadlock, that is, in each reachable state a transition is possible. This can be specified by the two following UPPAAL properties:

a. No deadlock:
```
A[] not deadlock
```

b. Context establishment:
```
(h1 != h2 and heuristics[h1][h2]) -->
  Context(h1,h2).established and Context(h2,h1).established
```

The first property, which checks if the model cannot deadlock, contains a predefined expression `deadlock`. The second property is not complete. It only verifies that if host `h1` is triggered to establish a Shim6 context for host `h2`, both contexts must become established. The actual property we would like to verify should apply to all host identifiers:

```
forall(h1:HostType) forall(h2:HostType) (
  (h1 != h2 and heuristics[h1][h2]) -->
  (Context(h1,h2).established and Context(h2,h1).established)
)
```

This property can be used with a network with a variable number of connected hosts. Unfortunately, the grammar for the requirement specification language does not allow this property. Quantifiers like `forall` and `exists` can only be applied to expressions, and not to property operators like `A[]` and `-->`. If we could prove that all instances of `h1` and `h2` are symmetric, then it is sufficient to check only one query. Fortunately, `h1` and `h2` are declared as `HostType`, meaning that they are scalar sets that are fully symmetric. This means that this single query is enough to verify the property for all host identifiers. However, UPPAAL does not accept concrete values of scalar types. This is solved by initializing two host identifiers `h1` and `h2`. This way, we can be sure that it is the same instance of `h1` and `h2` on both sides of the property operator.

Next to the deadlock property, we would like to verify the REAP model for failure detection and locator pair exploration. The first property states that if a link fails that was in use by the current address pair, the link failure will eventually be detected if payload is being transmitted. The second property states that if such a link failure is detected, the host will eventually find a new working locator pair if no vital network problem has occured. That means that the host will reach the Operational state, or the host is no longer able to send packets (`P1`), or the host is no longer able to receive packets (`P2`):

a. Failure detection:
```
(sending_payload and
 failed[REAP(h1).curr_local][REAP(h1).curr_peer]) -->
exists(h2:HostType) REAP(h2).Exploring
```
b. Locator pair exploration:
```
REAP(h1).Exploring --> (REAP(h1).Operational or P1 or P2)
```

where
```
P1 = forall(i1:IPv6Type) forall(i2:IPv6Type)
       ((UseIP[i1] == h1 and UseIP[i2] != h1 and
         i1 != ipv6_nil and i2 != ipv6_nil)
       imply (failed[i1][i2]))

P2 = forall(i1:IPv6Type) forall(i2:IPv6Type)
       ((UseIP[i1] == h1 and UseIP[i2] != h1 and
         i1 != ipv6_nil and i2 != ipv6_nil)
       imply (failed[i2][i1]))
```

4.1 Abstractions

The model described in chapter 3 is closely related to the definition in the protocol draft. Consequently, the model became too big to verify nontrivial properties. This forces us to reduce the state space by performing several abstractions.

Symmetry reduction During the creation of the full Shim6 model, attention was payed to symmetry reduction [Dil93]. The previous chapter showed that hosts can be modeled symmetrically. After all, it doesn't matter which host acts as initiator or responder. With Shim6, all hosts are equally related, unlike a situation where one host acts as master and the other hosts act as slaves. Symmetry also applies to the `Network` automata. Host identifiers and network identifiers can be defined symmetrically and this helps to reduce the space and memory consumption of Uppaal .

For host automata and network automata it is relatively easy to see why they can be modeled symmetrically. We would like to use this technique to reduce even more state space. For example, can IP addresses be treated symmetrically? Normally, the answer would be no. That is because some IP addresses are allocated for special purposes. For example, an IPv6 address beginning with `FE80` is considered a link-local address and should never be routed. There are no local addresses in the Shim6 model, only unicast global addresses are considered. We do need to have

the special unicast unspecified address (denoted as `::`). If the unspecified address was unnecessary, all addresses were global unicast addresses and could be modeled symmetrically.

Section 3.1.5 showed that IP addresses still can treated symmetrically. This is realized with the use of some modeling tricks. A special instance `ipv6_nil` of `IPv6Type` is declared to be the unspecified address. All other places in the model that deal with specified IPv6 addresses `ip` are provided with additional checks `ip!=ipv6_nil`. Assignment and identity testing are still the only operations used on IPv6 addresses. At this point, the model has symmetric IPv6 unicast global addresses, including the unicast unspecified address. The same trick can be applied to context tags and nonces.

Decomposition If only certain properties of a system are interesting, it is often possible to reduce the system into a smaller system in which the property still holds. This is also true for our model. Since we are interested in Shim6 enabled hosts, we can abstract from hosts that do not recognize the Shim6 extension headers. In other words, our model can abstract from ICMP error messages and NO-SUPPORT states. Removing all variables and locations that relate to this behavior, leads to a smaller number of locations and transitions, thus also to the number of reachable states. Another applied decomposition is context forking. Because context forking is not considered, all FII variables can be safely removed.

Abstraction The full model simulates quite realistic the protocol specification at the cost of state space. For example, our packet structure follows the message formats in the Shim6 draft. However, many elements return in different message formats. Nonces need to be repeated and context tags need to be exchanged. The packet can also be implemented as one C-struct, containing only the necessary elements:

```
typedef struct {
        IPv6Type src;
        IPv6Type dest;
        Type type;
        ContextTagType ct1; // for common ct
        ContextTagType ct2; // for r1bis ct
        NonceType n1; // for request, initiator nonce
        NonceType n2; // for responder nonce
        IPv6Type ULID_src;
        IPv6Type ULID_dest;
        bool Ls[IPv6Type];
        ValidatorType v_resp;
} ipv6_packet;
```

Now only ten variables are used for the packet structure. The variable `Ls` is an array of IPv6 addresses. The validator `v_resp` is a structure containing six variables. So instead of the $44 + 3m$ packet variables in the full model, now only $9 + 6 + m$ packet variables per automaton need to be maintained, m being the number of IPv6 addresses in the model. Another abstraction was made with respect to the locations `i2sent` and `i2bissent`. These two locations almost have the same behavior. The two locations could be merged. Basically, the only difference between the two states is that they retransmit different messages. In order to be able to merge the locations,

the function `send_i2` needs to be modified. In the abstracted model, the function first checks the state of the context before constructing the packet. Logically, an I2bis message is constructed if the state was I2BIS-SENT , otherwise an I2 message is constructed.

Dead variable reduction Dead variable reduction is a technique that reduces state space by assigning values to variables that are currently not used [Yor00]. A variable is considered used if in a transition it appears in the guard or in the right side of an assignment. A variable is defined if it appears in the left side of an transition. A variable is considered *dead* at a location if on every possible transition the variable is defined before it is used or it is not used at all. For example, the variable `tries` in `Config(i,tag)` is said to be dead in location `established`. Assigning `tries=0` upon occurrence of a transition that leads to `established`, does not affect any of the relevant properties. We can also identify many dead variables in the `context` structure. These can be assigned an appropriate value in the specific context functions.

4.2 Results

4.2.1 Context establishment

For two network automata and two hosts with each two IP addresses and a nonce maximum of 3, both properties can now be verified within one minute. Where the first property (no deadlock) satisfies immediately, the model needs some adjustments before the second property can be satisfied. These adjustments indicate problems in the protocol description that need to be corrected to ensure a clear, non-ambiguous and correct description. The first three problems were found during the verification process. The other were found during the modeling of Shim6.

4.2.1.1 Receiving payload packets in I2-SENT or I2BIS-SENT

According to the protocol description, Shim6 should send a R2 message if a payload packet was received and the corresponding context is in the state I2-SENT or I2BIS-SENT. Reason for this is that the R2 might be lost during transmission. However, it is the other end point that needs to retransmit a R2 message, since that host is already in the ESTABLISHED state, but this host is still waiting to complete the establishment. In order to complete the process, it should trigger the other end point to retransmit the R2 message by retransmitting the I2 or I2bis message (depending on the location). This can be considered an error in the draft, since with this description the two hosts might never become established.

4.2.1.2 Deadlock in I2-SENT or I2-BISSENT

A deadlock may occur in I2-SENT or I2BIS-SENT because the draft makes a wrong assumption about retransmissions. Retransmissions of I2 and I2bis messages are

not considered a requirement. The retransmissions are considered truly optional. The draft also states that retransmitted messages might be rejected after a certain amount of time, so it is recommended to fallback to I1-SENT after a certain number of retries. A problem arises if a developer decides not to implement the retransmissions. Consider that if the I2 or I2BIS message was not accepted in the first time or was lost during transmission, it would also not receive a reply. The implementation decides not to retransmit messages and therefore does not fallback to I1-SENT, resulting in a deadlock scenario. My suggestion to solve this problem is to fallback after a certain amount of time, instead of after a number of retransmissions.

4.2.1.3 Updating context clock

The draft was unclear about when to update the context clock. This clock is used to check that the context is still in use. The document suggests in chapter 9, that this clock needs to be reset every time a Shim6 payload is transmitted or received. Actually, the authors of the draft meant that the clock needed to be updated every time a payload triggered a context lookup[1]. This is necessary, because payloads do not include a Shim6 extension header before a locator switch. However, the host must update the context clock, in case of a locator switch just after the context was torn down. Verification pointed out that the context clock also needs to be updated when receiving a valid Shim6 control message. If this does not occur, strange situations might occur where one host tears down its context when the other host is just becomes established.

4.2.1.4 Confusion about context variables

The protocol draft is not clear about which context variables to maintain. In chapter 6, a conceptual data structure is given. Also a table is provided, that shows which context variables need to be stored during the different context states. But this table also introduces new elements to the structure, that were not discussed earlier. For example, the table mentions the initiator and responder nonces. It seems that they did not describe this in the conceptual model because nonces are transient information that are only used during context establishment. The other described context elements are stable context information. The same applies for the newly introduced R1bis context tag, which is solely used for context recovery. In this case, the table omitted some context elements, as the verification showed that the responder nonce should also be stored in I2-SENT and the initiator nonce, responder nonce and validator should also be stored in I2BIS-SENT. This also means that the responder nonce and validator in the I2 and I2bis message should not be copied from the triggering message, but should be retrieved from the context variables. This is a rather subtle difference with respect to retransmitting messages.

1. This was clarified on the Shim6 mailing list

4.2.1.5 Nonce counter for host acting as responder

When a host receives an I2 or I2bis message, it should check the contained nonce and validator to determine that the message was not replayed. Because a responder may not create state when generating the R1 or R1bis validator, the nonce and validator should be verified with global host information, for example a host's clock. According to the draft, the idea then is that the responder has only a Secret `S` and a responder nonce counter that he is incrementing each time it receives a new I1. In order to verify that the message is a recent one, the host only accepts messages that contain nonces not older than `VALIDATOR_MIN_LIFETIME`. But in order to make this verification, it will need to store the time each nonce was generated. This implies context specific state which is not allowed. An alternative as suggested by the authors of the draft is to use the system clock value as responder nonce, hashed with some secret value. This way, each message can be verified with the system clock and `VALIDATOR_MIN_LIFETIME` only.

4.2.1.6 Generating and validating the responder validator

A responder validator needs to be generated for R1 and R1bis messages. These validators need to be validated on receipt of I2 and I2bis messages. The draft is however inconsistent with generating and validating these values. For generation, different elements are used than for validation. It is not difficult to see that validating a hash value using alternate elements will fail.

4.2.2 Reachability protocol

For one network automata and two hosts with each two multiple IP addresses, the REAP verification properties can now be verified within one minute. Unfortunately, I was not able to verify the system with more network automata, because this significantly increased the number of states. This means that we only can verify the properties assuming that there are no race conditions. Still, two minor issues were revealed.

4.2.2.1 Failure detection

The first encountered problem is about receiving the last payload packet. The problem is illustrated in Fig. 4.1. Consider two hosts A and B using the address pair (A1, B1) and (B1, A1) respectively. There is unidirectional payload traffic (P_1) from A to B. Host B makes the communication bidirectional by replying with keepalive messages (K_1). It will send multiple keepalives in response to one single payload packet, because it needs to send keepalives at certain intervals. Suddenly the link for address pair (A1, B1) fails. Host A sends its final payload packet (P_2) but because the link is not operational anymore, it is not delivered. Host B continues sending its keepalive messages until the keepalive timeout is reached. Because A now receives K_1 messages, it does not detect that the link (A1, B1) failed at the end, and that the message P_2 was lost.

If a connection-oriented, reliable protocol is running on top of Shim6, which is usually the case, this is not a problem. The upper layer protocol guarantees reliable and in-order delivery of data from sender to receiver, and will detect that P_2 was lost. The send timer in REAP is restarted and the link failure will eventually be detected. However, if you running an unreliable protocol on top of Shim6, the message is lost without being noticed.

(1) $A \rightarrow B : P_1$
(2) $B \rightarrow A : K_1$
(3) $B \rightarrow A : K_1$
Link (A1,B1) failure
(4) $A \rightarrow : P_2$
(5) $B \rightarrow A : K_1$

Figure 4.1: Failed failure detection

4.2.2.2 Locator pair exploration

When a host suspects a link failure, it sets its reachability state to Exploring. The Shim6 proposes that four initial probes are transmitted when in the exploration phase. Retransmissions should occur sequentially and with exponential backoff, starting the first retransmission after 0.5 seconds. The interval of the retransmissions may not grow beyond the limit of 60 seconds. This limits the number of addresses that can in practice be used for multihoming, because the upper layer protocol will fail if the exploration takes too long. According to the REAP draft, "It will continue sending at this rate until a suitable response is triggered or the SHIM6 context is garbage collected, because upper layer protocols using the SHIM6 context in question are no longer attempting to send packets." [page 14, section 4.3]. With this approach, after more than five minutes, only fifteen different locator pairs can be tried. Suppose our multihoming hosts A and B each maintain five locators. The number of possible locator pairs for A to reach B is 25. In case that currently only one of the uplinks for A is working, there are only five working locator pairs. And because within five minutes, ten address pairs of host A will never be probed and have become useless. This means that host A has to rely on the default address selection to pick one of the five working locator pairs within fifteen tries. In reality, even less probes can be tried, since the upper layer protocol will probably time out within two minutes. You can imagine, that a large multihomed site with hundreds of IPv6 addresses even run a bigger risk not picking the right locator pair in time.

Our model is not able to simulate that many IPv6 addresses per host, so we adapt the number of initial probes to just one. Also, no exponential backoff is used. Still, the above sketched problem occurred. Perhaps, if the default address selection algorithm is smart enough to directly pick the working locator pair candidates, the problem does not occur. But this would not be a realistic approach, since default address selection will need to collect information about the network performance. Still, we would like to verify the behavior in this best case scenario. Therefore, I have made our selection procedure very efficient by letting it only select working locator pairs:

```
bool default_address_selection(IPv6Type src, IPv6Type dst) {
        if (src == ipv6_nil or dst == ipv6_nil)
          return false;
        if (UseIP[src] != i or UseIP[dst] == i)
          return false;

        if (all_failed())
          return candidate[src][dst];

        if (failed[src][dst])
          return false;
}
```

If the complete network is down, the selection procedure falls back to the original candidate selection. Otherwise, it decides its candidates with use of the array `failed`. In this best case scenario, the property of locator pair exploration is satisfied.

4.3 Formalization results

Modeling Shim6 gave insight of the importance of formal methods. The process of formalization of the protocol description already revealed a number of problems. Furthermore, UPPAAL turned out to be a sufficient tool to model parts of the Shim6 protocol. Shim6 needs to deal with timing constraints, which can be simulated in UPPAAL . Furthermore, UPPAAL uses C-like syntax, that is recognizable for implementers of kernel applications. UPPAAL provides a graphical syntax for finite state machines, which makes it easy to understand for protocol designers. In short, the tool has a low threshold access and is able to model non-trivial real time systems.

Still, improvements to UPPAAL can be made, especially to the C-like syntax. The following issues were revealed:

- The C-like syntax is sometimes a bit too primitive. For Shim6, each message format can be followed by one of multiple type specific data types. Unfortunately, it is not possible to point to the corresponding data type. Instead, we had to include all data types into the IPv6 packet. This creates a lot of unnecessary state. With a C-like `union` type or pointers, we were able to reduce this.
- With Shim6, the IPv6 addresses in the locator lists can be updated, but also the length of the locator lists may differ. However, UPPAAL does not allow variable length of lists. We need to initialize our locator lists with the maximum length, that is, the total number of IPv6 addresses.
- The context establishment is actually a four-way communication exchange. Incoming messages require outgoing messages in the form of a reply. Ideally, we would like synchronization channels that can handle input and output messages in one transition. This way, we could model the "receiving of I1 and sending of R1" in one transition. However, input / output transitions are not possible in UPPAAL . We can only provide the input part by assigning values in the assignment section of the input transition. For receiving and sending,

we need to model at least two transitions.

- Our model needed initialization, in order to assign IPv6 addresses to hosts. We introduced a new automaton just for this initialization. Adding a new automaton for this seems a bit roundabout.
- The grammar for the requirement specification language does not allow the application of quantifiers like `forall` and `exists` to property operators like `A[]` and `-->`. For Shim6, this has the consequence that the properties become unclear.

Chapter 5

Conformance testing

Shim6 is close to become a proposed standard. Because of the high interest, several code bases are currently working on implementations of Shim6. The aim is to become experienced with Shim6 and to detect flaws. At the moment, one Shim6 beta implementation is available (UCL, [Bar06]). If this implementation is conform the draft, it should encounter the same problems that have been revealed by UPPAAL .

Checking if the implementation is conform the specification is however not that simple. Therefore, a relation between the implementation and the informal language in the draft must be examined. In order to facilitate this process, the draft uses keywords to determine the requirements for implementations. We will test the UCL implementation against some of the informal notated requirements in our created, virtual environment.

5.1 Test setting

Three components are needed to setup the test environment. First, a Shim6 simulator that is able to speak with the UCL implementation is needed. This tool is used to simulate the problems that were detected by the verifier and trigger the requirements as described in the draft. Second, a virtual network should be used as test environment. It is recommended that a new kernel is tested in a virtualized environment before adapt the kernel physically. Our test environment contains three virtual hosts: two Shim6 enabled hosts and one running the simulator. Finally, a traffic analyzing tool is needed to monitor the context establishment exchange. This allows us to examine and verify the behavior of the implementation.

5.1.1 The shimulator

Our ad hoc Shim6 simulation code, which has been called the shimulator, consists of two programs. One program intercepts incoming and outgoing Shim6 messages

on a certain interface. It maintains the context state and determines its actions depending on the state and the incoming messages. These actions are implemented according to the Shim6 draft. The other program allows you to manually inject your own Shim6 packets via the command line. This part of the shimulator may be used to create special Shim6 packets, for example with unknown critical options. The responses to these packets will again be captured by the interceptor program.

The interceptor program, called the shimterceptor, can be called with multiple interfaces:

```
# ./shimterceptor eth0 eth1 eth2
```

Both programs must be run as root, because you need to have privileges to access the interface. The shimterceptor will first construct its locator set with the IPv6 addresses that are assigned to the provided interfaces. In this case, the locator set contains three locators. After constructing the set, the shimterceptor will listen to the first provided interface, in this case `eth0`. The shimterceptor will only be terminated by way of user input. Whenever a packet is passing by this interface, the function handler `intercept` is executed. This determines if the packet is a Shim6 packet by checking the next header field of the IPv6 header data, and if the packet is an incoming or outgoing packet by checking the source and destination fields. If the message is outgoing and the message type is equal to I1, a context is allocated. If the message is incoming, the actions follow the rules of the draft.

The other program, called the shimjector, is also called with multiple interfaces. Again, these are needed to construct the locator set. The shimjector needs more parameters. For instance, it needs to know what the destination of the packet is, which type of message needs to be injected, and if options must be included.

```
# ./shimjector -send I2 foo.nlnetlabs.nl eth1 eth0 eth2 -fc -uc -l
```

In this example, an I2 control message is transmitted to the computer foo.nlnetlabs.nl via interface `eth1`. The message will contain a critical forked instance identifier option (`-fc`), an critical unknown option (`-uc`) and a non-critical locator list option (`-l`). The locator list option will contain three locators in the order they were provided through the command line.

5.1.2 The virtual network

For the test environment, VMWare software is used. VMWare Workstation is a tool that supports virtualization on x86 architecture machines [Ada06]. It allows many different kinds of operating systems and provides three kinds of networking:

- **Bridged Networking** treats guest machines as a unique identity on the real network, unrelated to its host.
- **Network Address Translation (NAT)-based Networking** lets guest machines share the IP and MAC addresses of the host. The virtual machine and the host share a single end point identifier. The virtual machine can access other end points in the external network, but reversed this does not work: External end points cannot initiate connections to the virtual machine.
- **Host-only Networking** configures the guest machine to allow network access only to the host.

The last type of networking is sufficient for our test environment. With host-only networking, virtual hosts are able to communicate without traffic being routed over the Internet. Data packets will not leave the host, though the network traffic is visible to the host computer. It is configured a virtual Ethernet adapter *vmnet1*, that is used to transmit the virtual traffic. The virtual traffic can be with a traffic analyzer, capturing packets on the host virtual adapter. Three guest hosts are connected to *vmnet1*. Two of them implement the UCL implementation of Shim6, version 0.4. The third runs our shimulator. The addresses for these machines are provided by the VMWare DHCP Server. This setup is schematically illustrated in Fig. 5.1.

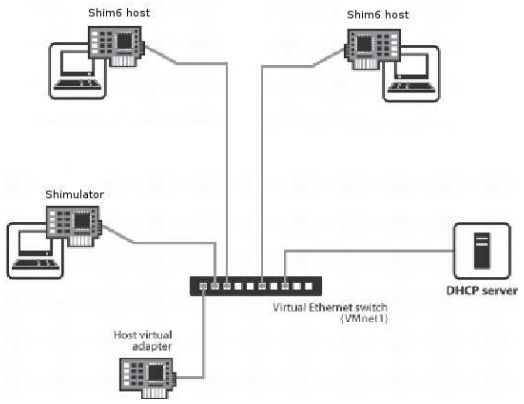

Figure 5.1: Setup of the conformance test environment

5.1.3 The Wireshark traffic analyzer

Wireshark [Lam07] (formerly known as Ethereal) is an open source network protocol analyzer. It captures messages on the virtual Ethernet adapter by setting it in promiscuous mode. Packets are than dissected and the pieces of information are shown to the user. Many protocols are supported as a result of people contributing that support. For my thesis, I have added support for the Shim6 protocol.

Shim6 is considered an IPv6 extension header, so in order to parse Shim6 packets, the code for sniffing IPv6 packets needs to be adapted. I shall discuss the most important modifications. A captured packet is provided to the dissect functions. First of all, the Shim6 header must be identified and structured. If the IPv6 packet is followed by Shim6 data, its next header field must be equal to 61:

```
#define IP_PROTO_SHIM6 61
```

The Shim6 header structure looks a lot like the other extension headers. It also starts with a next header field and a header extension length. This header contains one extra bit `P` to distinguish control messages from payload packets:

```
struct ip6_shim {
        guint8  ip6s_nxt; /* next header */
        guint8  ip6s_len; /* header extension length */
        guint8  ip6s_p;   /* P field and first 7 bits of remainder */
};
```

The protocol needs to be registered:

```
{ &hf_ipv6_shim6,
  { "SHIM6 ", "ipv6.shim6", FT_NONE, BASE_NONE, NULL, 0x0, "", HFILL }
}
```

This allows you to filter Shim6 traffic. Several other registrations are made for all Shim6 elements. This is useful to filter, for example, only Shim6 payload messages. Now, the dissecting code for Shim6 can be added. This code is called when the function `dissect_ipv6` reads out a next header indicating Shim6:

```
case IP_PROTO_SHIM6:
        shim6 = TRUE;
        advance = dissect_shim6(tvb, offset, tree);
        ...
```

The captured packet may contain several different protocol message formats. A Shim6 payload message will typically contains an Ethernet header, an IPv6 header, a Shim6 header, a TCP header and some application header. Each different header is represented as a tree. The function `dissect_shim6` adds a new subtree to the IPv6 tree. First, the three common element, the next header field, the header extension length and the `P` bit, are added to the Shim6 subtree:

```
ti = proto_tree_add_item(tree, hf_ipv6_shim6, tvb, offset, len, FALSE);
shim_tree = proto_item_add_subtree(ti, ett_ipv6);

/* Next Header */
proto_tree_add_uint_format(shim_tree, hf_ipv6_shim6_nxt, tvb,
   offset + offsetof(struct ip6_shim, ip6s_nxt), 1, shim.ip6s_nxt,
   "Next header: %s (0x%02x)", ipprotostr(shim.ip6s_nxt),
   shim.ip6s_nxt);

/* Header Extension Length */
proto_tree_add_uint_format(shim_tree, hf_ipv6_shim6_len, tvb,
   offset + offsetof(struct ip6_shim, ip6s_len), 1, shim.ip6s_len,
   "Header Ext Length: %u (%d bytes)", shim.ip6s_len, len);

/* P Field */
proto_tree_add_boolean(shim_tree, hf_ipv6_shim6_p, tvb,
   offset + offsetof(struct ip6_shim, ip6s_p), 1,
   shim.ip6s_p & SHIM6_BITMASK_P);
```

If `shim.ip6s_p & SHIM6_BITMASK_P` is verified, the captured packet is identified as a Shim6 payload. The receiver context tag field is added to the tree and dissecting continues with the data that follows Shim6. Otherwise, the packet is a Shim6 control message. The type of message is retrieved, as well as the used protocol (HIP or Shim6), the checksum and the type specific data:

```
/* Message Type */
ctype = val_to_str(shim.ip6s_p & SHIM6_BITMASK_TYPE, shimctrlvals,
   "Unknown Message Type");
proto_tree_add_uint_format(shim_tree, hf_ipv6_shim6_type, tvb,
   offset + offsetof(struct ip6_shim, ip6s_p), 1,
```

```
   shim.ip6s_p & SHIM6_BITMASK_TYPE, "Message Type: %s", ctype);

/* Protocol bit (Must be zero for SHIM6) */
proto_tree_add_boolean_format(shim_tree, hf_ipv6_shim6_proto, tvb, p,
   1, tvb_get_guint8(tvb, p) & SHIM6_BITMASK_PROTOCOL, "Protocol: %s",
   tvb_get_guint8(tvb, p) & SHIM6_BITMASK_PROTOCOL ? "HIP" : "SHIM6");
p++;

/* Checksum */
csum = shim_checksum(tvb_get_ptr(tvb, offset, len), len);

if (csum == 0) {
        proto_tree_add_uint_format(shim_tree, hf_ipv6_shim6_csum, tvb,
           p, 2, tvb_get_ntohs(tvb, p), "Checksum: 0x%04x [correct]",
           tvb_get_ntohs(tvb, p));
} else {
        proto_tree_add_uint_format(shim_tree, hf_ipv6_shim6_csum, tvb,
           p, 2, tvb_get_ntohs(tvb, p),
           "Checksum: 0x%04x [incorrect: should be 0x%04x]",
           tvb_get_ntohs(tvb, p),
           in_cksum_shouldbe(tvb_get_ntohs(tvb, p), csum));
}
p += 2;

/* Type specific data */
advance = dissect_shimctrl(tvb, p, shim.ip6s_p & SHIM6_BITMASK_TYPE,
   shim_tree);
```

The function `dissect_shimctrl` takes care of the dissecting of control messages. After these formats, some options may be included. In theory, there may be an infinite number of options. The end of the Shim6 header and thus also the options is indicated by `offset+len`. As long as our variable `p`, which is used as a marker that registers where we have left off. As long as `p` does not pass by the end of the Shim6 header:

```
if (p < offset+len && shim_tree) {
        ti_shim = proto_tree_add_text(shim_tree, tvb, p, len-(advance+6),
           "Options (%u bytes)", len-(advance+6));

        opt_tree = proto_item_add_subtree(ti_shim, ett_ipv6);
        while (p < offset+len) {
                p += dissect_shimopts(tvb, p, opt_tree);
        }
}
```

This ends the dissection of the Shim6 packet data. A screenshot of a typical context establishment exchange is shown in Fig. 5.2.

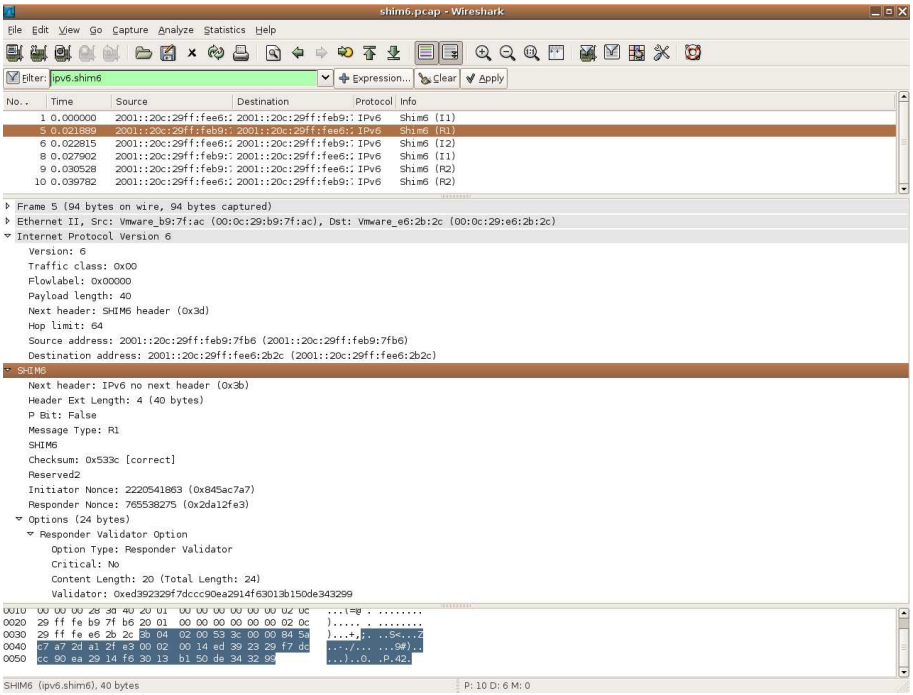

Figure 5.2: Screenshot of Wireshark that followed a Shim6 context establishment exchange. The R1 message is shown in detail. The filter expression ensures that only Shim6 packets are shown. At the foot of the picture you can see a number of selected bytes that represent the Shim6 data of the packet.

5.2 Results

Setting up the test environment was not as straight-forwarded as expected. First of all, the Shim6 implementation seemed to crash the host every time the context was established. The implementation required a daemon `reapd` to be active. This needed to be started manually or at start up of the host. If `reapd` was not running, the implementation would freeze the kernel[1].

The other problem is that the implementation did not accept link local addresses to the locator sets. Our virtual hosts only had link local addresses, which resulted in exchanging empty locator lists to each other. Link local addresses were not accepted due to technical problems. In the context of Shim6, link local addresses only make sense if they are associated with a link. In the current protocol version, this additional information cannot be stored.

After solving the first two encountered problems, we let the two Shim6 implemen-

1. This was fixed in the version 0.4.3 of the UCL implementation, released on 17 April 2007.

tations communicate with each other. Because both implementations are identical, both hosts decide to set up a context at the same time. This results in a concurrent context establishment with crossing I1 messages. This seems to work fine. Next, the implementation is tested against the requirements.

5.2.1 Implementation requirements

The draft uses different kinds of keywords to describe the implementation requirements. The keyword *MUST* is used to identify an absolute requirement. *MUST NOT* means that the described behavior is absolutely prohibited. The keyword *SHOULD* may only be ignored if there exists a valid reason not to implement it, but the full implications must be understood and carefully weighed. The keyword *SHOULD NOT* indicates the opposite behavior. Finally, the keyword *MAY* means that the described behavior is truly optional.

The exact meaning and the use of these keywords are defined by the IETF [Bra97], but Shim6 is not consistent in using these keywords. For example, the draft says [page 64, section 7.13]:

> *The host moves to ESTABLISHED state.*

This presumably means that the host *MUST* move to ESTABLISHED state. Also, behavior is described using conflicting keywords. The draft describes the receipt of an option that the host does not understand [page 41, section 5.14]:

> *If C=1 then the host SHOULD send back an ICMP parameter problem (type 4, code 1), with the Pointer referencing the first octet in the option Type field.*

When describing the receipt of Shim6 control messages, unknown critical options are mentioned again [page 81, section 12.2]:

> *If there is any option in the message with C=1 that isn't known to the host, then the host MUST send an ICMPv6 Parameter Problem, with the Pointer field referencing the first octet of the Option Type.*

In this case, the keywords *SHOULD* and *MUST* are used to identify different kind of requirement levels for two identical behavioral situations. Nevertheless, we will observe the Shim6 requirements. These are copied from the draft specification [Nor06].

5.2.1.1 Renumbering implications

When a host is renumbered, one or more of its locators become invalid and possibly new locators are added to its locator set. This enables the possibility that communication for an ULID pair continues after one or both ULID addresses have become unavailable. The unavailable ULID may be reassigned to another site while it is still being used for the existing communication. The Shim6 protocol is not intended to let communication survive renumbering, but this potential source for confusion can be avoided if we require that any communication using a ULID *MUST* be terminated when the ULID becomes invalid. This can be accomplished by discarding the ULID context.

For this requirement, we let the two Shim6 enabled hosts communicate with each other. When the context is established, we deactivate the interface `eth0` associated with the ULID pair of the initiator. This should result in a removal of the context. Instead, the hosts start probing for a new locator pair. We may discuss whether deactivating an interface is a form of renumbering. It does meet the description "one ore more of its locators become invalid". On the other hand, it may also be seen as link failure. In this case, the implementation acts correctly by starting a probe exploration process.

If another interface `eth1` is deactivated, a new probe exploration starts. It is expected to find the final locator associated with interface `eth2`. This is true, but the implementation does not recognize this. Ten seconds (equal to the send timeout) after sending an probe with the state field set to Operational, the host will send a new probe with the state field set to Exploring. This process repeats itself continuously. Barre already discussed the exploration process termination on the Shim6 mailing list, but this looks like a different problem. The exploration may start only if the send timeout expires. According to the change log of the latest Shim6 release of UCL, we may assume that this had something to do with the probe reports:

> *The probe report lists are cleared upon send timer expiry. This avoids pretending we are reachable through a path that has recently failed (and maybe caused the send timer expiry)*

5.2.1.2 The upper layer

In the I1, I2 and I2bis messages, when the IPv6 source and destination addresses in the IPv6 header does not match the ULID pair, the ULID pair option *MUST* be included. In particular, if a ULID pair option was included in the I1 message then it *MUST* be included in the I2 message as well. If the ULP is using separate contexts, the message associated with the forked context *MUST* include the Forked Instance Identifier option carrying the instance identifier value for this context.

The UCL implementation does not yet cover context forking. Also, the including of an ULID pair option is too difficult to trigger. Therefore, we should try to trigger the implementation to send an I1 for an ULID pair, with non-equal IPv6 addresses. Normally, the ULID pair is also used for being the source and destination fields [page 6, section 1.3]:

> *Underneath, and transparently, the multihoming shim selects working locator pairs with the initial locator pair being the ULID pair... If communication subsequently fails the shim can test and select alternate locators. A subsequent section discusses the issues when the selected ULID is not initially working hence there is a need to switch locators up front.*

But because no locator set is known yet before Shim6 establishes the context, this behavior is hard to trigger.

5.2.1.3 Critical options

Options may be critical. If the critical bit `C` is equal to one, the parameter *MUST* be recognized by the recipient. All the control messages can contain any options with

`C=0`. If there is any option in the message with `C=1` that isn't known to the host, then the host *MUST/SHOULD* send an ICMPv6 Parameter Problem, with the Pointer field referencing the first octet of the Option Type. Such messages *MUST NOT* be processed.

For this test, we use our shimulator and send packet containing critical and non-critical options. There turned out to be no behavioral difference between sending critical and non-critical options. So setting the critical option does not force options to be recognized or, in case of unknown options, be discarded. There were some other issues found when executing these tests:

- The locator list option should include internal padding in the content length field. The implementation does not include the padding, resulting in an incorrect retrieval of the locator list. The locators that are stored are bogus IPv6 addresses, as shown in Fig. 5.3. Technically, the implementation does not overrule requirements from the draft. The draft should use a keyword to require that internal padding is included in the content length field.
- An I1 message including locator preferences, CGA PDS, CGA signature, FII, ULID pair or unknown options is being processed, but if the same options are included in an I2 message, the message is silently dropped or does not pass the verifications.
- The ULID pair is not taken from the ULID pair option if it is present.
- The first element of the locator list is assigned to be the locator pair, while this initially should be the ULID pair (see Fig. 5.3).

5.2.1.4 Locator preferences

The locator preferences option is used to express performance, priority and weight values for each locator. Every value takes up one octet in the message format. If all three values are used, the locator preferences are stored in three octets per locator. A field called the element length enables the host to retrieve how many values are used per locator. The draft does not specify the format when the element length is more than three, except that any such formats *MUST* be defined so that the first three octets are the same as in the above case, that is, one octet flags field followed by one octet priority field, and one octet weight field.

When a locator preferences option is included in the I2 message, the packet is not being processed. That it, no reply is sent back. If an update request message or an I2bis message is transmitted, the Shim6 enabled host replies with an ICMP parameter problem. This indicates that update messages and context recovery are not yet implemented.

5.2.1.5 Context confusion

As part of establishing a new context, each host has to assign a unique context tag. Since the payload extension headers are demultiplexed based on the context tag value only, the context tag *MUST* be unique for each context. In addition, in order to minimize the reuse of context tags, the host *SHOULD* randomly cycle through the 2^{47} context tag values.

```
State: established
Peer ulid: 2001:0:0:0:20c:29ff:fe37:b076
Local ulid: 2001:0:0:0:20c:29ff:fee6:2b2c
FII: 0
Peer locators list :
        0:0:20:100:0:0:2:c29
        fffe:37b0:7620:100:0:0:2:c29
Local locators list :
        2001:0:0:0:20c:29ff:fee6:2b36   refcnt=2
        2001:0:0:0:20c:29ff:fee6:2b40   refcnt=2
        2001:0:0:0:20c:29ff:fee6:2b2c   refcnt=3
(debug) shim6_nb_glob_locs=3
Peer context tag : 264731363814842
Local context tag : 07
Init nonce : 0
Age : 78 seconds
Preferred peer locator : 0:0:20:100:0:0:2:c29
Preferred local locator : 2001:0:0:0:20c:29ff:fee6:2b2c
refcnt : 1
REAP data :
        state : operational
        timer information :
                keepalive timer currently inactive
                send timer currently inactive
```

Figure 5.3: The Shim6 context after a context establishment exchange between the UCL implementation and our shimulator. The locator list has been retrieved incorrectly. The locator pair is not initially set to be the ULID pair.

Context confusion might occur after context loss or premature garbage collection. The confusion can be detected when an I2, I2bis or R2 message is received, because it is required that those messages *MUST* include a sufficiently large set of locators in the locator list option. The host can determine whether or not two contexts are maintained for the same peer by comparing if there is any common locators in the locator set of the peer. Thus, when receiving such a message, a host *MUST* look for possible context confusion. This *MUST* be done before a possible R2 message is replied.

If the situation occurs, the context *MUST* be removed: it *MUST NOT* be used anymore to send any packets. It *MAY* attempt to re-establish the old context by sending a new I1 message and moving its state to I1-SENT. In any case, the host *MUST NOT* keep two contexts with overlapping peer locator sets and the same context tag in ESTABLISHED state, since this would result in demultiplexing problems on the peer.

First of all, the UCL implementation increments the context tag each time a new context is allocated. This provides unique context tags, but does not meet the requirement to cycle through the 2^{47} possible context tag values. Second, the Shim6 draft does not specify what can be considered a sufficiently large set of locators. In our test setting, a locator list option contain three valid locators at maximum. Nevertheless, we try to trigger context confusion. A host can detect confusion when it receives an I2, I2bis or R2 message. Context confusion is usually the consequence of (premature) context loss. In order to be able to simulate context loss, we first let the UCL implementation and the shimulator set up a context. At this point in time, both hosts maintain an established context. We may trigger context recovery by sending a new I1 message [page 57, section 7.5]:

> *If one end has garbage collected or lost the context state, it might try to create a new context state (for the same ULID pair), by sending an I1 message. The peer (that still has the context state) will reply with an R1 message and*

the full 4-way exchange will be performed again

An I1 packet with the same ULID pair, but a different context tag is included. The R1 message is delivered to the established context state associated with the host that implements Shim6, since I1 packets are dispatched upon their ULID pair and the FII. If the state exists in ESTABLISHED state and the locators do fall in the sets, then the host compares the context tag for the context with the context tag contained in the I1 message. If they do not match, a R1 message must be sent back. However, the UCL implementation incorrectly considers the I1 message to be of the same context, and a R2 message is replied. Context confusion was not triggered, but we did reveal another issue.

5.2.1.6 Locator verification

HBA/CGA verification *SHOULD* be performed by the host before the host acknowledges the new locator, by sending an Update Acknowledgment message, or an R2 message. Before a host can use a locator (different than the ULID) as the destination locator it *MUST* perform HBA/CGA verification if this was not performed before upon receipt of the locator set. In addition, it *MUST* verify that the ULID is indeed present at that locator. This verification is performed by doing a return-routeability test as part of the Probe sub-protocol. If the verification method in the locator list option is not supported by the host, or if the verification method is not consistent with the CGA Parameter Data Structure, then the host *MUST* ignore the locator list and the message in which it is contained, and the host *SHOULD* generate an ICMP parameter problem (type 4, code 0), with the Pointer referencing the octet in the verification method that was found inconsistent.

To test if the locators are verified at the right moments, the shimulator sets up a context with one Shim6 enabled host. In the I2 message, a locator list option is provided that includes the locators of the shimulator, a multicast address and a locator that belongs to the second Shim6 enabled host. Unfortunately, HBA or CGA is not yet supported by the UCL version of Shim6. The implementation stores all provided peer locators in the peer locator list, as illustrated inf Fig. 5.4. Because HBA and CGA is not yet supported, none of the above requirements regarding HBA and CGA are satisfied.

When the host needs to start locator exploration, the implementation does verify the peer addresses: Only valid, global unicast addresses are used for probing. This means that also the second, not-participating Shim6 host is probed (at address 2001::20c:29ff:feb9:7fb6). If this host implemented context recovery, it should reply with a R1bis message. Now, the probe is silently discarded.

5.2.1.7 Receiving messages

When receiving Shim6 control messages, the host *MUST* verify that:

- The Shim6 checksum field is correct.
- The Shim6 header length does not claim to end past the end of the IPv6 packet.

```
State: established
Peer ulid: 2001:0:0:0:20c:29ff:fe37:b076
Local ulid: 2001:0:0:0:20c:29ff:fee6:2b2c
FII: 0
Peer locators list :
        2001:0:0:0:20c:29ff:fe37:b076
        2001:0:0:0:20c:29ff:fe37:b080
        ff02:0:0:0:0:0:0:1
        2001:0:0:0:20c:29ff:feb9:7fb6
Local locators list :
        2001:0:0:0:20c:29ff:fee6:2b40   refcnt=2
        2001:0:0:0:20c:29ff:fee6:2b36   refcnt=2
        2001:0:0:0:20c:29ff:fee6:2b2c   refcnt=3
(debug) shim6_nb_glob_locs=3
Peer context tag : 264731363814842
Local context tag : 030
Init nonce : 0
Age : 8 seconds
Preferred peer locator : 2001:0:0:0:20c:29ff:fe37:b076
Preferred local locator : 2001:0:0:0:20c:29ff:fee6:2b2c
refcnt : 1
REAP data :
        state : operational
        timer information :
                keepalive timer currently inactive
                send timer currently inactive
```

Figure 5.4: The Shim6 context after a context establishment exchange between the UCL implementation and our shimulator. The peer locators are stored without any verification.

- the Shim6 header length does not claim that the Shim6 packet is smaller than the minimum Shim6 packet length.
- Neither the IPv6 source field nor the destination field is a multicast address.

When receiving I2 or I2bis messages, the host verifies that the responder nonce is a recent one. Nonces that are no older than VALIDATOR_MIN_LIFETIME *SHOULD* be considered recent. If a CGA Parameter Data Structure (PDS) is included in the message, then the host *MUST* verify if the actual PDS contained in the message corresponds to the peer ULID. If none of the verifications failed, the host looks for a corresponding context. In case the I2/I2bis message leads to an update of the context, the host *MUST* send a R2 message back. Before updating the peer locator set, the host *SHOULD* perform the HBA/CGA validation for the peer locator set.

Also upon the receipt of a R2 message including a CGA PDS option, the host *MUST* verify that the actual PDS contained in the message corresponds to the peer ULID.

When receiving update requests or acknowledgments, the host *MUST* verify that the IPv6 source address field is part of peer locator set and that the IPv6 destination address field is part of local locator set. If this is not the case, the sender of the update message has a stale context which happens to match the local context tag for this context. In this case the host *MUST* send a R1bis message, and otherwise ignore the update message. Again, if a CGA PDS is included, the host *MUST* verify if the actual PDS contained in the packet corresponds to the peer ULID.

In this test case, the shimulator sends various Shim6 packets that do and do not meet these requirements. The implementation all respectively processed and discarded them correctly. Because locator verification and update messages are not implemented yet, the other requirements could not be tested.

5.2.1.8 Retransmissions

If the initiator does not receive an R2 message after I2_TIMEOUT time after sending an I2 message it *MAY* retransmit the I2 message. The responder validator option might have a limited lifetime, that is, the peer might reject responder validator options that are older than VALIDATOR_MIN_LIFETIME to avoid replay attacks. Thus, the initiator *SHOULD* fall back to retransmitting the I1 message when there is no R2 received after retransmitting the I2 message I2_RETRIES_MAX times.

The same behavior applies to retransmitting I2bis messages. Retransmissions were revealed by UPPAAL to be incorrect behavior, which we also like to test against the UCL implementation. Therefore, we trigger the shimmed host to initiate a context establishment exchange. The shimulator will reply with R1, but will not answer to the I2 message. This way, the context of the initiator will stay in I2-SENT. Wireshark confirms that the implementation does retransmit I2 messages and fallback occurs after sending four retries, and it will not deadlock in I2-SENT.

The roles are changed: the shimulator now acts as initiator and the Shim6 host acts as responder. To verify that the Shim6 host rejects responder validators that are older than VALIDATOR_MIN_LIFETIME, the shimulator waits thirty seconds before sending the I2 message. This equals the maximum lifetime of the responder validator (Actually, VALIDATOR_MAX_LIFETIME would be a better name for this). After thirty seconds, the I2 message still is replied with a R2 message. There might be a small deflection in the time units of the hosts, so we try the test again. But this time the shimulator waits for forty five seconds. With this setup, the I2 message is discarded.

5.2.1.9 Sending payload

When sending packets, if there is a ULID pair context for the ULID pair, and the ULID pair is no longer used as the locator pair, then the sender needs to transform the packet. First, the IP address fields are replaced. The IPv6 source address field is set to the local locator and the destination address field is set to the peer locator. This *MUST NOT* cause any recalculation of the ULP checksums, since the ULP checksums are carried end-to-end and the ULP pseudo-header contains the ULIDs which are preserved end-to-end.

For this test, the shimulator does not have to be used. We start a communication between the two shimmed hosts. Context establishment occurs and the payload transmission continues without modifications to the packets. If we check the ULP checksum, in this case the TCP checksum, Wireshark confirms that this checksum is correct. We trigger locator pair exploration by deactivating one of the interfaces. Payload will be tagged with a Shim6 extension header. Now, Wireshark complains that the TCP checksum is incorrect. This is expected, since the TCP checksum was not recalculated, but the packet data was modified. The packet is restored at the network layer. On that level, the TCP checksum will match the packet data again.

5.2.1.10 Receiving payload

When receiving packets, the context looks if there exists a corresponding context. If no context is found, the receiver *SHOULD* generate a R1bis message. According to the draft, the host should reply with a R2 message if the context is in the state I2-SENT of I2BIS-SENT. Formal verification showed that this was incorrect behavior. Instead of a R2 message, the host must reply an I2 or I2bis message, depending on the context state.

Since the implementation does not yet support context recovery, no R1bis message is generated when the shimulator sends a Shim6 payload. Next, we will trigger the shimmed host to initiate a context establishment exchange, just like we did with the retransmissions test case. Instead of finalizing the exchange by sending R2, the shimulator immediately starts sending Shim6 tagged payload. The implementation follows the incorrect description of the draft and replies with R2, while it should reply with an I2.

5.2.1.11 Message formats

Shim6 messages must follow certain message formats. Many messages contain bits that are reserved for future use. They *MUST* be ignored on receipt. Several options may be included in the message. The length of an option is a multiple of eight bytes. When needed, padding *MUST* be added to the end of the parameter so that the total length becomes a multiple of eight bytes. If padding is added, the option length field *MUST NOT* include the padding.

Special care is needed for the locator list option. When it is sent, the necessary HBA/CGA information for verifying the locator list *MUST* also be included. The subset of locators included in the correspondent Locator List Option which verification method is set to CGA. When using a CGA signature for the locators, the signature *MUST* include the locators in the order they are listed in the locator list option.

For all transmitted messages by the shimmed host, the message formats were conform the requirements. The implementation does not include HBA or CGA information for verifying the locator list, but this is because support for locator verification is not yet implemented.

5.2.1.12 Context teardown

It is *RECOMMENDED* that hosts do not tear down the context when they know that there is some upper layer protocol that might use the context. It is also *RECOMMENDED* that implementations minimize premature teardown by observing the amount of traffic that is sent and received using the context, and only after it appears quiescent, tear down the state. A reasonable approach would be not to tear down a context until at least five minutes have passed since the last message was sent or received using the context.

The implementation maintains an Age counter, which is reset every time the context is used. The context is teared down a long time after five minutes, probably because this longer lifetime facilitates in the testing phase.

5.2.2 Test conclusions

The test results gave us insight of the quality of the implementation in question. Some of the requirements were satisfied, for example correct message formats, no ULP checksum re-calculation and correct retransmissions. Some of the requirements were not satisfied, for example no difference between critical and non-critical options and incorrect actions when receiving payload. Testing the requirements also revealed some other issues, for example omitted internal padding in the content length field and wrong assignment of the initial locator pair.

The tests described in section 5.2.1.8 and section 5.2.1.10 were able to reproduce the issues described in section 4.2.1.2 and section 4.2.1.1 respectively. This stresses out the importance of formal methods, as being a complementary technique to verify properties of Internet protocols. The other tests encountered other issues that sometimes because the draft described incorrect behavior, and sometimes because the draft was misinterpret. Unfortunately, many requirements could not yet be tested, simply because the behavior was not yet implemented.

Chapter 6

Conclusions

Shim6 provides a scalable solution specific for multihoming while minimizing deployment disruption. Currently, the protocol is still in development. This thesis shows some efforts that try to improve the quality of the protocol specification. The process of formalization, verification and testing are shown to be successful contribution methods. Formalization was useful in clarifying the protocol specifications. It revealed some ambiguities and unclarities in the draft specification. Verification also revealed some errors. These issues have been communicated to the authors of the Shim6 draft. They have been acknowledged and adjusted in the specification.

The technique of model checking has been used for formalization and verification of the Shim6 protocol. A great advantage of model checking is that it is intended to find many errors quickly and automatically. The Uppaal model checker was used to formalize the Shim6 specifications. Two critical parts of Shim6, the context establishment and REAP, have been formalized and verified with Uppaal . This tool benefits from its possibility to model timing constraints and its rich syntax. However, model checking has to deal with the state space explosion problem. Also, the syntax could still be improved with more C-like data types. Furthermore, the verification process has shown that the requirement specification language is still somewhat primitive.

One Shim6 beta implementation has been tested on conformance. Therefore, I have added support for capturing Shim6 packets to the Wireshark traffic analyzer. This can be useful for other implementors and testers of Shim6. The conformance test has been an incomplete process, because we had to deal with an unfinished beta implementation and a draft specification written in informal language. Besides, the draft showed inconsistency in its requirements. This makes it hard to verify if an implementation is conform the standard, and this conformance test cannot been be seen as a waterproof test. Nevertheless, several problems were revealed. Some problems were already discovered by the model checker, some were discovered by interoperability between our Shim6 simulator and the Shim6 implementation and some were discovered by conformance testing. These problems are reported to the author of the implementation and this may facilitate in the Shim6 standard creation process.

6.1 Future Work

This thesis opens possibilities for future research of the formalization and implementation of Shim6.

Formalization Our current Shim6 model is divided in two UPPAAL systems, to avoid state space explosion. Also, properties were verified with a very low number of hosts, network automata and nonce values. Furthermore, the Shim6 documentation has developed in the last year. Our model is receptive for several adaptations that may lead to improvements. The most important betterments would be:

- Integrating the REAP model with the context establishment model, to provide a complete picture of Shim6.
- Processing the changes in the draft, in order to keep the models up to date.
- Expanding the model with behavior that was omitted, for example context forking and HBA/CGA.
- Applying adaptations that may lead to a more efficient state space, making the verification process faster.

This can be realized by modifying the model, or by collaboration with the UPPAAL team to improve the quality in general. For the latter, several suggestions for improving the model checker have been provided, like adding input/output transitions and extending the C-like syntax and the grammar for the requirements specifications.

Implementation Several code bases are already working on Shim6 implementations. Currently, only one beta implementation for Linux 2.6 is available. If Shim6 wants to be an Internet standard, more implementations are needed that are complete, that can operate in different systems and that interoperate with other implementations. Lots of work is needed to include HBA/CGA locator verification, context recovery and context forking. Also, the draft specification is still open for improvements. The Shim6 work group is still discussing the Shim6 architecture and location with respect to IPSec and fragmentation, the use of Shim6 error messages, calculation of the ULP checksum at middleboxes and many other issues. In general, the IETF is discussing the multihoming architecture, whether Shim6 is the right solution. Another problem that has not yet been mentioned is context scalability. Since multihoming creates context per remote communicating host, it is important to know how many contexts can be maintained at the same time before the local host runs out of memory. Suppose a large site like YouTube, that processes hundred millions of video streams a day, becomes multihomed with Shim6. With such large numbers, context scalability becomes an issue.

Appendix A

List of abbreviations

BGP-4 *Border Gateway Protocol version 4*
A routing information protocol used to announce routes between autonomous systems, for example between service providers.

BU message *Binding Update message*
A message used to annouce mappings between HoA and Coa addresses in MIPv6.

CGA *Cryptographically Generated Addresses*
A form of IPv6 address where the interface identifier is derived from a cryptographic hash of the public key.

CoA *Care-of Address*
An IP address serving as locator for a mobile node in MIPv6.

DHCP *Dynamic Host Configuration Protocol*
A protocol used by computers to request and obtain IP addresses and other addressing information from a DHCP server.

FII *Forked Instance Identifier*
A number to identify forked Shim6 contexts, in order to handle context forking.

FSM *Finite State Machine*
A model of behavior composed of a finite number of states, transitions between those states, and actions.

HBA *Hash Based Addresses*
A form of IPv6 address where the interface identifier is derived from a cryptographic hash of all the prefixes assigned to the host.

HIP *Host Identity Protocol*
A method of separating the end-point identifier and locator roles of IP addresses, realizing strong authentication between hosts at TCP/IP stack level.

HoA *Home Address*
An IP address serving as a stable end point identifier for a mobile node in MIPv6.

ICMP *Internet Control Message Protocol*
An extension to IP, allowing the generation of error messages, test packets and informational messages related to IP.

IEEE *Institute of Electrical and Electronics Engineers*
An organization composed of engineers, scientists, and students, best known for developing standards for the computer and electronics industry.

IETF *Internet Engineering Task Force*
A large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet.

IESG *Internet Engineering Steering Group*
A committee formed to help the IETF chair.

IP *Internet Protocol*
A data-oriented protocol that is encapsulated in the network stack of a computer, used for communicating data across packet-switched networks.

IPSec *IP Security*
A suite of protocols for securing IP communications by means of authentication and encryption.

IPv4 *Internet Protocol version 4*
Version 4 of the Internet Protocol.

IPv6 *Internet Protocol version 6*
Version 6 of the Internet Protocol.

MIPv6 *Mobile IPv6*
An extension to the IPv6 standard to support Internet connection for wireless devices, such as mobile phones or notebooks. MIPv6 preserves communication between nodes that move around in the network.

NAT *Network Address Translation*
A technique in which IP addresses are rewritten as they pass through a router or firewall, enabling a local-area network (LAN) to use one set of IP addresses for internal traffic.

OSI model *Open Systems Interconnection Basic Reference Model*
An abstract description for computer network protocol and network stack design.

PA space *Provider Assigned address space*
A block of IP addresses assigned to an end-user by a service provider.

PI space *Provider Independent address space*
A block of IP addresses assigned to an end-user directly by a Registry, without going through an service provider.

PDS *Parameter Data Structure*
The information that CGA and HBA exchanges in order to inform the peer of how the interface identifier was computed.

REAP *Reachability Protocol*
The mechanisms and protocol messages to achieve failure detection and locator pair exploration.

RFC *Request For Comments*
A document describing an informal Internet standard.

RIR *Regional Internet Registry*
An organization overseeing the allocation and registration of Internet number resources within a region.

Shim6 *Site Multihoming by IPv6 Intermediation*
A network layer protocol for providing IPv6 multihoming, without the use of PI space.

SYN *Synchronize*
A type of packet used by TCP.

TCP *Transmission Control Protocol*
A transport layer protocol that transmits multiple packet data between applications. TCP is a reliable, connection-oriented protocol that enables two hosts to establish a connection and exchange streams of data.

TCP/IP stack *Transmission Control Protocol/Internet Protocol stack*
A set of communications protocols that implements the protocol stack on which the Internet runs.

UCL *Universite catholique de Louvain*
University of Louvain, a code base for Shim6.

UDP *User Datagram Protocol*
A transport layer protocol that transmits multiple packet data between applications. UDP is an unreliable, connectionless protocol that, unlike TCP, does not guarantee packet delivery.

ULID *Upper Layer Identifier*
An IP address serving as a stable end point identifier for a Shim6 enabled host.

ULP *Upper Layer Protocol*
A protocol running on top of IP.

Appendix B

Adjustments to the Shim6 model

B.1 Updated context establishment

The context establishment model did not satisfy the properties and some adjustments had to be made. These modifications are also made retroactively in the full context establishment model. First of all, retransmitting I2 and I2bis messages is no longer optional. This means that the loop transitions in locations `i2sent` and `i2bissent` that resets the clock `y` are removed, as illustrated in Fig. B.1.

Updating the context clock and and the actions triggered when receiving payload had to be changed in the function `contextlookup` in the declarations of the Dispatcher. When receiving payload and the context is in location `i2sent` or `i2bissent`, the host does not longer reply with R2. Updating the context clock now happens every time the context is consulted, so it moves to the top of the function body:

```
void contextlookup() {
        send = NO_SHIM;
        goto = NOWHERE;

        peer = UseIP[p.src];
        ctx_clock[i][peer] = 0; // moved to the top

        // section 12.1
       if (p.shim6.type == PAYLOAD) {
                if (p.shim6.payload.ct_recv != ct_nil and p.shim6.payload.ct_recv == ctx_CT1[i][peer]) {
                        if (ctx_state[i][peer] == I2SENT)
                                send = I2; // send I2 instead of R2
                        else if (ctx_state[i][peer] == I2BISSENT)
                                send = I2bis; // send I2bis instead of R2
                        else if (ctx_state[i][peer] == EFAILED or ctx_state[i][peer] == NOSUPPORT)
                                send = R1bis;
                }
                else if (ctx_state[i][peer] == IDLE or ctx_state[i][peer] == EFAILED or
                                                       ctx_state[i][peer] == NOSUPPORT)
                        send = R1bis;// generate R1bis
     }

     ...
}
```

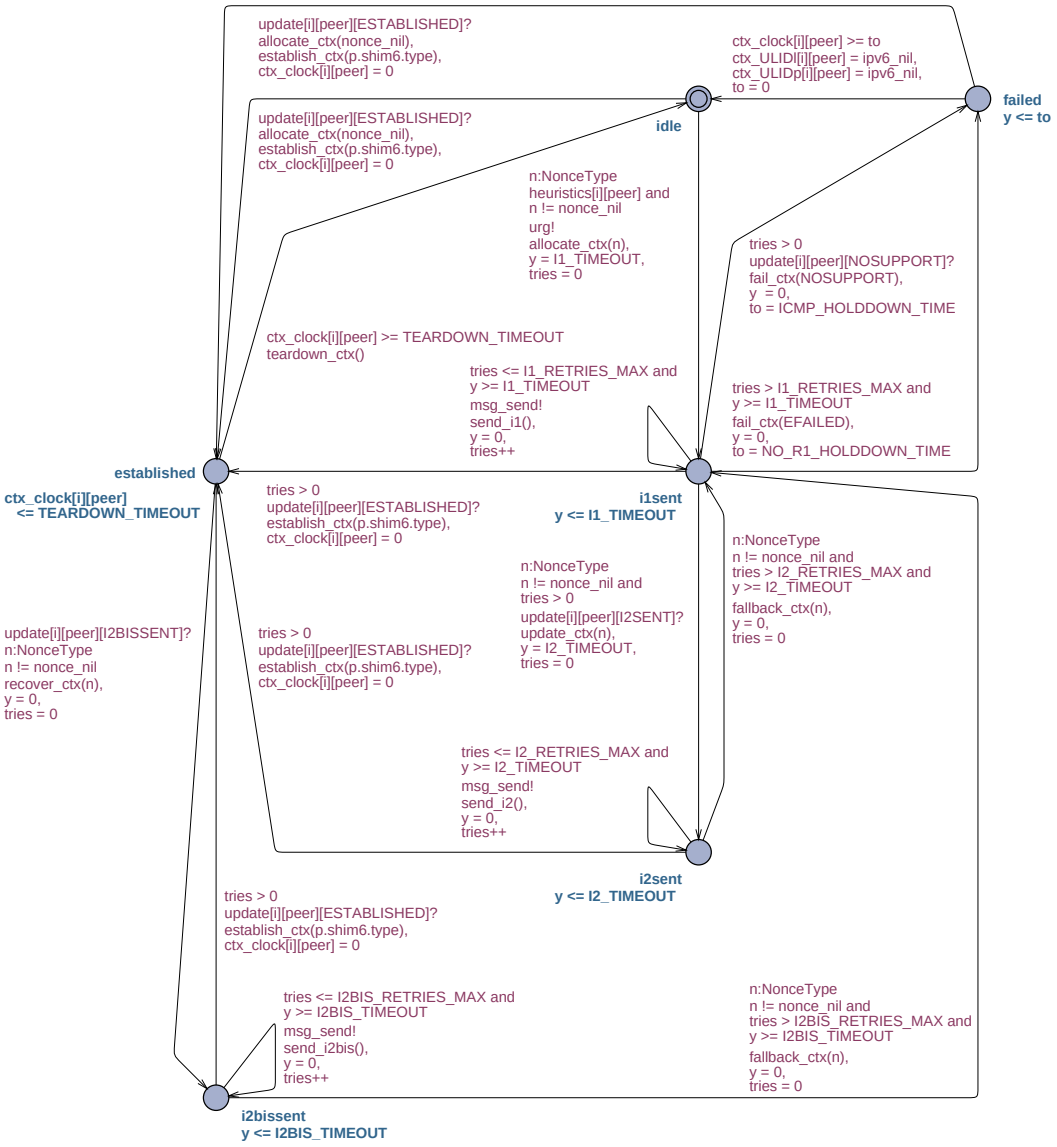

Figure B.1: The updated automaton `Context[i][peer]`

B.2 Abstracted context establishment

This section describes the abstractions that have been applied to the model, so that verification can be fulfilled.

B.2.1 Packet structure

The full model follows the message formats in the draft. For every message format, a new structure is defined. Because many elements return in different message formats, the packet can also be implemented as one C-struct:

```
typedef struct {
        IPv6Type src;
        IPv6Type dest;
        Type type;
        ContextTagType ct1; // for common ct
        ContextTagType ct2; // for r1bis ct
        NonceType n1;   // for request, initiator nonce
        NonceType n2;   // for responder nonce
        IPv6Type ULID_src;
        IPv6Type ULID_dest;
        bool Ls[IPv6Type];
        ValidatorType v_resp;
} ipv6_packet;
```

Instead of the $44 + 3m$ packet variables in the full model, now only $9 + 6 + m$ packet variables per automaton need to be maintained, m being the number of IPv6 addresses in the model. This obscures the reality of the model, but no behavioral changes have been made. This also simplifies our functions that clear packet data:

```
void vzero(ValidatorType &vt) {
        vt.ct_init = ct_nil;
        vt.n_resp = nonce_nil;
        vt.ULID_src = ipv6_nil;
        vt.ULID_dest = ipv6_nil;
        vt.Lp_src = ipv6_nil;
        vt.Lp_dest = ipv6_nil;
}

void pzero() {
        p.src = ipv6_nil;
        p.dest = ipv6_nil;
        p.type = NO_SHIM;
        p.ct1 = ct_nil;
        p.ct2 = ct_nil;
        p.n1 = nonce_nil;
        p.n2 = nonce_nil;
        p.ULID_src = ipv6_nil;
        p.ULID_dest = ipv6_nil;

        vzero(p.v_resp);

        for(i:IPv6Type)
                p.Ls[i] = false;
}
```

B.2.2 Merging locations i2sent and i2bissent

The location `i2bissent` bears resemblance to location `i2sent`. Basically, the only difference between the two states is that they retransmit different messages. In order to be able to merge the locations, the function `send_i2` needs to be modified:

```
void send_i2() {
        pzero();

        p.src = ctx_Lpl[i][peer];
        p.dest = ctx_Lpp[i][peer];

        if (i2b) {
                ctx_state[i][peer] = I2BISSENT;
                p.type = I2bis;
                p.ct2 = ctx_CTr1bis[i][peer];
        } else {
                ctx_state[i][peer] = I2SENT;
                p.type = I2;
        }

        p.ct1 = ctx_CTl[i][peer];
        p.n1 = ctx_noi[i][peer];
        p.n2 = ctx_nor[i][peer];
        p.v_resp = ctx_var[i][peer];
        if (ctx_ULIDl[i][peer] != ctx_Lpl[i][peer] or ctx_ULIDp[i][peer] != ctx_Lpp[i][peer]) {
                p.ULID_src = ctx_ULIDl[i][peer];
                p.ULID_dest = ctx_ULIDp[i][peer];
        }
        for (j:IPv6Type)
               p.Ls[j] = ctx_Lsl[i][peer][j];
}
```

Fig. B.2 shows how the locations have been merged. The following constants can be removed from the global declarations:

```
const int I2BIS_TIMEOUT = 4; // seconds
const int I2BIS_RETRIES_MAX = 2;
```

B.2.3 All Shim6 enabled hosts

In the model, only Shim6 enabled hosts participate. This means that a context will never move to the location `failed` by means of no Shim6 support at the remote host. This can be verified by adding a boolean `nosupp` that is set `true` only if the "no support" transition is taken. The property `A[] not nosupp` will now be satisfied. This means that this transition can safely be removed, as well as all constant values and variables that deal with no support detection. The abstracted `Context` automaton, illustrated in Fig. B.2, does not behave differently than the full context establishment model.

The following constants can be removed from the global declarations:

```
const int ICMP_HOLDDOWN_TIME = 600; // seconds
const int NOSUPPORT = 7;
const int ICMP = 10;
```

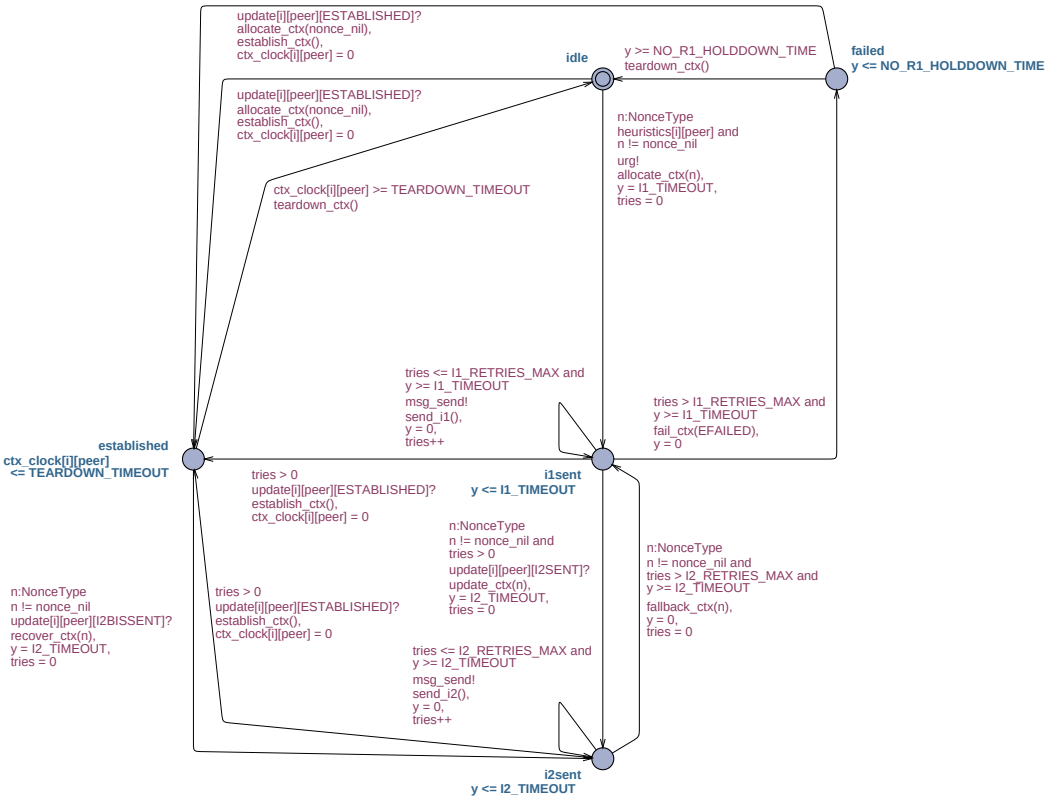

Figure B.2: The abstracted automaton `Context[i][peer]`

B.2.4 No context forking

Context forking is not yet widely discussed in the draft, and it was not considered in the full model. However, it did provide variables to expand the model with this functionality. The model satisfies the property:

```
A[] forall(h1:HostType) forall(h2:HostType) ctx_FII[h1][h2] == 0
```

proving that no forked contexts exist. All variables that consider context forking can be safely removed.

Bibliography

[Abl03] Abley, J. and Black B. and Gill, V. *Goals for IPv6 Site-Multihoming Architectures*, August 2003.

[Abl05] Abley, J. and Lindqvist, K. and Davies, E. and Black B. and Gill, V. *IPv4 Multihoming Practices and Limitations*, July 2005.

[Ada06] Adams, K. and Agesen, O. A comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13. ACM Press, October 2006.

[Alu94] Alur, R. and Dill, A. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Ark06] Arkko, J. and Beijnum, I. van. *Failure Detection and Locator Pair Exploration Protocol for IPv6 Multihoming*, December 2006.

[Aur05] Aura, T. *Cryptographically Generated Addresses (CGA)*, March 2005.

[Bag05] Bagnulo, M. *Hash Based Addresses (HBA)*, October 2005.

[Bar06] Barre, S. and Bonaventure, O. Developpement d'extensions au Kernel Linux pour supporter le multihoming IPv6. Technical report, Universite Catholique de Louvain, June 2006.

[Beh04] Behrmann, G. and David, A. and Larsen, K.G. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editor, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[Bra96] Bradner, S. *The Internet Standards Process – Revision 3*, October 1996.

[Bra97] Bradner, S. *Key words for use in RFCs to Indicate Requirement Levels*, March 1997.

[Car96] Carpenter, B. and Rekhter, Y. *Renumbering Needs Work*, February 1996.

[Cla91] Clark, D. and Chapin, L. and Cerf, V. and Braden, R. and Hobby, R. *Towards the Future Internet Architecture*, December 1991.

[Con98] Conta, A. and Deering, S. *Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*, December 1998.

[Dee98] Deering, S. and Hinden, R. *Internet Protocol, Version 6 (IPv6) Specification*, December 1998.

[Dil93] Dill, D.L. and Ip, C.N. Better Verification Through Symmetry. In D. Agnew, L. Claesen and R. Camposano, editor, *Computer Hardware Description Languages and their Applications*, number 32 in A, pages 87–100,

Ottawa, Canada, April 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[Dra03] Draves, R. *Default Address Selection for Internet Protocol version 6 (IPv6)*, February 2003.

[Edd06] Eddy, W. *TCP SYN Flooding Attacks and Common Mitigations*, December 2006.

[Hen03] Hendriks, M. and Behrmann, G. and Larsen, K.G. and Niebert, P. and Vaandrager, F.W. Adding symmetry reduction to uppaal. In Kim Gulstrand Larsen and Peter Niebert, editor, *FORMATS*, number 2791, pages 46–59. Springer, 2003.

[Hen06] Henderson, T. and Ahrenholz, J. OpenHIP 0.4.1. Technical report, Internet Engineering Task Force, Internet Research Task Force, December 2006.

[Joh04] Johnson, D. and Perkins, C. and Arkko, J. *Mobility support in IPv6*, January 2004.

[Lam07] Lamping, U. Wireshark Developer's Guide. Technical report, Wireshark, 2007.

[Mey06] Meyer, D. and Zhang, L. and Fall, K. *Report from the IAB Workshop on Routing and Adressing*, December 2006.

[Nor06] Nordmark, E. and Bagnulo, M. *Level 3 multihoming shim protocol*, May 2006.

[Rek06] Rekhter, Y. and Li, T. and Hares, S. *A Border Gateway Protocol (BGP-4)*, January 2006.

[Sav05] Savola P. and Chown T. A Survey of IPv6 Site Multihoming Proposals. In *Proceedings of the 8th International Conference of Telecommunications (ConTEL 2005)*, pages 41–48, June 2005.

[Tae06] Taewan, Y. SHIM6 Implementation. Technical report, Electronic Telecommunications Researching Institute, July 2006.

[van03] van Langevelde, I. and Romijn, J. and Goga, N. Founding FireWire bridges through Promela prototyping. In *Parallel and Distributed Processing Symposium, 2003*, pages 307–320, April 2003.

[Yor00] Yorav, K. Exploiting Syntactic Structure for Automatic Verification. *PhD thesis*, June 2000.