FORMAL ANALYSIS OF ROUTING PROTOCOLS

Davor Obradovic

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2002

Supervisor of Dissertation

Val Tannen
Graduate Group Chair

Acknowledgements

Chapters 3, 4, 5 are based on joint work with Carl Gunter and Karthikeyan Bhargavan who I have been privileged to have as colleagues. Their help was of crucial importance for this thesis. The work presented in Chapter 6 extends the research ideas from [20, 19, 21, 16, 17] by Timothy G. Griffin, F. Bruce Shepherd, Gordon Wilfong, Lixin Gao and Jennifer Rexford. I wish to thank them for the inspiration they provided. I would like to thank the CISCO Corporation, and especially Alvaro Retana, for supporting my research on BGP. I would also like to thank Rajeev Alur, Dimosthenis Anthomelidis, Alwyn Goodloe, Roch Guerin, Sampath Kannan, Pankaj Kakkar, Sanjeev Khanna, Moonjoo Kim, Insup Lee, Michael McDougall, Oleg Sokolsky, Carolyn Talcott and Mahesh Viswanathan for their comments and support.

Abstract

FORMAL ANALYSIS OF ROUTING PROTOCOLS

Davor Obradovic

Supervisor:

The task of a routing protocol is to discover and maintain paths between distant points in a network. We study the problem of formal correctness analysis of such protocols. Traditionally, routing protocols have been evaluated by testing. While testing provides useful insights about feasibility of the protocol, its interaction with the environment and generally about the average case behavior, it can not provide guarantees which would limit the worst case behavior. Our formal correctness analysis differs from testing in several aspects:

- It focuses on a protocol standard, rather than a particular implementation.
- It considers all possible behaviors, rather than just the average case.
- It is formal, in the sense that we prove mathematical theorems about routing protocols.
- It can detect errors which are not visible as performance degradation.

We carried out three case studies, each involving a different routing protocol:

1. A well known distance-vector routing protocol RIP. We proved convergence towards optimal routes, together with a sharp real-time bound for convergence time.
2. A recent routing protocol AODV for mobile "ad hoc" networks. We identified flaws that can lead to loop formation, suggested modifications and proved that the modified protocol is loop free.
3. Currently the only widespread inter-domain routing protocol BGP. We first establish a timeless measure for the speed of convergence. We then refine it into a real-time model of the protocol, which we use to prove a general theorem about an upper bound on convergence time. Finally, we show several practical corollaries of the theorem.

Contents

Acknowledgements **ii**

Abstract **iii**

1 Introduction **1**

2 Routing Protocols **5**

2.1 Routing in IP Networks 5

2.2 Routing in Ad Hoc Networks 7

2.3 Convergence of Routing Protocols 8

2.4 Summary 9

3 Methodology **10**

3.1 Verification Attributes of Routing Protocols 10

3.2 Formal Methods 11

3.2.1 Applications of Formal Methods 15

3.3 Verification Methodology 17

3.3.1 Standard 18

3.3.2 Pseudo-code 19

3.3.3 Specification 20

3.3.4 Requirements 23

3.3.5 Verification 25

3.4 Summary 28

4 Routing Information Protocol (RIP) **29**

4.1 Protocol Description 29

4.2 Standard vs. Theory 31

4.3 Convergence of RIP . . . 32
4.3.1 Proof Methodology . . . 35
4.4 Sharp Timing Bounds for RIP Stability . . . 36
4.4.1 Proof Methodology . . . 39
4.5 Summary . . . 40

5 Ad Hoc On Demand Distance Vector Protocol (AODV) 41
5.1 Protocol Description . . . 41
5.2 Loop Conditions . . . 44
5.3 Path Invariants and Loop Freedom . . . 45
5.3.1 Proof Methodology . . . 49
5.4 Summary . . . 50

6 Border Gateway Protocol (BGP) 51
6.1 Introduction . . . 51
6.2 Protocol Description . . . 52
6.3 Convergence of BGP . . . 56
6.4 The Stable Paths Problem and the Simple Path Vector Protocol . . . 58
6.5 Measuring BGP Oscillation . . . 64
6.6 Real-time Model of BGP . . . 66
6.6.1 Consistency of Real-time Constraints . . . 76
6.7 Application: Delay-cost Consistent Policies . . . 77
6.8 Application: Convergence of SPVP . . . 80
6.9 Application: Safe SPVP . . . 81
6.10 Summary . . . 89

7 Conclusions 91

A Code Samples 94
A.1 RIP Pseudo-code . . . 94
A.2 AODV Pseudo-code . . . 96
A.3 AODV Modification . . . 100
A.4 SPVP Pseudo-code . . . 100

Bibliography 101

List of Tables

5.1 Protocol verification effort 50

List of Figures

2.1 An autonomous system . . . 5

3.1 LTL grammar . . . 23

4.1 Maximum convergence time . . . 38

5.1 Initial configuration . . . 45
5.2 Looping scenario . . . 45
5.3 Looping scenario . . . 46
5.4 Looping scenario . . . 46
5.5 Looping scenario . . . 47
5.6 Looping scenario . . . 47

6.1 UPDATE message . . . 53
6.2 An example of the usage of MED . . . 54
6.3 Examples of Stable Path Problems . . . 61
6.4 A potential dispute between policies . . . 63
6.5 Hierarchical topology . . . 82
6.6 Selective export rules . . . 83
6.7 A slow converging instance of safe SPVP. . . . 86

Chapter 1

Introduction

Routing protocols establish procedures for discovery and maintenance of paths between distant points in the Internet. This is a vital networking task. Errors in routing protocols can lead to connection outages which can be very costly, given the current volume of traffic flowing through the Internet. Because of that, it is important to verify, before a protocol is used, that its behavior will conform to the expectations. This analysis can be carried out along two major directions: *performance* and *correctness*.

Performance testing is a traditional way of analyzing routing protocols. It focuses on measuring performance in simulated or real networks. This approach provides crucial insights about feasibility of the protocol, its overhead, and the average-case behavior in general.

Nevertheless, the protocol design may hide errors which do not surface as performance degradation. To detect this kind of errors, one needs to perform a more systematic correctness analysis. There are two fundamental differences between performance and correctness analysis. Firstly, rather than focusing on the *implementation* (as with performance testing), correctness analysis focuses on proving properties about the protocol *standard*. Secondly, results of correctness analysis apply to all possible behaviors, rather than just the average-case.

These two approaches are not competing alternatives—rather, they supplement each other. For instance, correctness analysis can be used early to detect basic design errors and set reasonable expectations for later performance testing. On the other hand, regardless of how great are the properties that one proves about the standard, a protocol can not be guaranteed to properly work in a real network before it is tested. Therefore, a good protocol validation methodology should include both correctness and performance analysis.

In addition to these two "pure" approaches, there are also "hybrid" approaches which combine

the two. For instance, simulations created for performance analysis can be used to carry out correctness testing [8].

The work described in this thesis is about correctness analysis of routing protocols. Routing protocols represent a specific domain for correctness analysis. This domain poses specific challenges to modeling and verification. However, at the same time, its constraints allow for effective usage of specific verification strategies, which are not widely applicable in the general case. One class of examples are abstractions of the network topology. Below are some of the key aspects specific to correctness analysis of routing protocols:

- All routing protocols are n-party protocols, where each party runs a copy of the same process. Consequently, the state space is unbounded, but very symmetric. Because of that, abstractions are essential for any tool-based verification. For instance, we often view the system as consisting of only two components: one representing a single router and the other representing its environment.

- Route computations involving different destinations are non-interfering. This is why we can often assume that we are dealing with a single destination. This simplifies the verification and significantly reduces the state space.

- Each process is a reactive system that maintains local state. The state changes in response to event occurrence. Each event is observed by only one party.

- The processes are connected in a certain network topology, which determines communication ability. Topology can change and it is essentially the only "global state" of the system. However, each party has only local access to that global state, since a router knows only about its immediate neighbors. This strongly distributed state space together with localized events increases the modularity of the protocol design and simplifies verification.

- All protocol requirements are expressed only in terms of the local *state* (and not, for instance, in terms of the observed event sequence). In other words, local states subsume the event information, so there is no need to keep track of the event history.

Rather than developing a new verification theory/tool, our goal was to study the impact of the above aspects in practice, for correctness analysis of some real, concrete routing protocols.

Our analysis methodology proceeds through several steps. We start by transforming the protocol standard into pseudo-code—a more structured and precise form. For this purpose, we developed a pseudo-language called POPSCL, specifically tailored for routing protocols. We then proceed by

creating a formal model (i.e. *specification*) of the protocol. In the case of tool-based verification, the specification is given in the form of machine-readable code. The next step consists of formalizing properties representing protocol requirements in the same model. Verification is the final step, where we attempt to prove (or disprove by a counter-example) that the specification satisfies the requirements.

Our methodology has the following important properties:

- It is formal—we prove mathematical theorems about routing protocols.
- It is systematic—considers all possible behaviors.
- It is applicable early, as soon as the protocol is designed and before it is implemented and deployed.
- It is partially automated through the use of model checkers and theorem provers.

The work is organized around three case studies, each involving a different routing protocol. The three protocols differ widely in their age, intended application domain and environmental assumptions. Below we summarize our results in each of the case studies:

1. RIP is one of the first distance vector routing protocols. It is a mature, well known protocol based on the asynchronous distributed Bellman-Ford protocol described in [5]. The RIP standard includes additional features that are intended to improve the real time behavior. Our results account for these additions. We proved the convergence of the protocol, provided a sharp real time bound for it, and proved that the protocol finds optimal routes.

2. AODV is a recent, still evolving routing protocol for ad hoc mobile networks. AODV has a special mechanism that is supposed to prevent formation of routing loops. We identified flaws in earlier versions of the standard that can cause loops to be formed. We then suggested modifications to the protocol and proved that they successfully avoid loops. Our results have directly affected the subsequent versions of the standard.

3. BGP is the only widely deployed inter-domain routing protocol. Our primary focus was the speed of convergence of BGP. We first establish a timeless measure for the speed of convergence called the *oscillation index*. Then we refine it through a real-time model of the protocol. We use the real-time model to prove a general theorem about an upper bound on convergence time. Finally, we apply the theorem to derive important convergence properties for three classes of configurations.

The rest of the document is organized as follows:

- Chapter 2 gives an introductory overview of routing protocols.
- Chapter 3 provides a description of our analysis methodology.
- Chapter 4 describes the RIP case study.
- Chapter 5 describes the AODV case study.
- Chapter 6 describes the BGP case study.

Chapter 2

Routing Protocols

2.1 Routing in IP Networks

The Internet is broadly organized into collections of networks called *autonomous systems (AS's)*. An AS may, for instance, be the internetwork of a company, a university, or an Internet Service Provider (ISP). Each AS consists of a set of networks which are interconnected by routers. A router can be connected to a network through one of its *interfaces*. Connections between routers and networks define the *topology*. Figure 2.1 shows an AS with four networks (clouds) and four routers (black squares). Router $r1$ has interfaces $i1$, $i2$ and $i3$, which connect it to the networks $n1$, $n2$ and $n3$ respectively. Hosts $h1$ and $h2$ belong to the network $n1$. Routers are said to be *neighbors* if they have interfaces to a common network. In our example, all the routers are neighbors of $r1$, but $r2$ and $r4$ are not neighbors. The goal of the routers is to forward packets between hosts

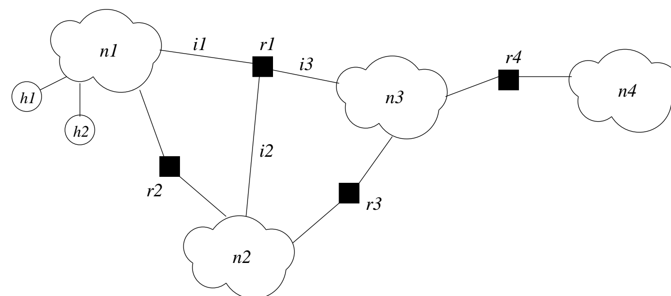

Figure 2.1: An autonomous system

(shown as circles) that are attached to the networks. When a packet arrives over one interface, a router has to decide, based on the destination address, which interface should it use to forward the

packet "closer" to its destination. In order to be able to make good forwarding decisions, routers need to maintain partial topology information in the *routing tables.* The aim of a routing protocol is to establish a procedure for updating these tables. In most cases, routing information can be exchanged only *locally* (i.e. between neighboring routers). However, the overall goal of a routing protocol is to establish good *global* paths across the network.

Routing in the Internet generally happens on two levels. On the *external* level, packets are routed to the border of their destination AS. After that, they are routed *internally* within that AS towards the destination. This is analogous to a highway system, where one first uses a fast express highway to get to the destination area and then uses slower roads with more exits to get exactly to the destination.

Routing protocols that are used for external routing between AS's are called *Exterior Gateway Protocols (EGPs)*, while those that are used for internal routing are called *Interior Gateway Protocols (IGPs).* The principal EGP is the Border Gateway Protocol (BGP). IGPs fall into two categories: distance vector routing and link state routing.

Distance vector protocols were among the first to be used in the Internet. In such protocols, each router maintains, for each destination, the name of an adjacent router that is (thought to be) one "hop" closer to the destination and an estimate of the number of hops to reach the destination. This information is periodically *advertised* to adjacent routers, and *updated* to take account of information from the advertisements of adjacent routers. The best-known protocol of this kind is RIP, which is still widely used because of its early inclusion in Unix operating systems. RIP is described in a series of IETF RFC's [23, 39, 36, 37, 38]. Another example of a distance vector protocol is the *Enhanced Interior Gateway Routing Protocol (EIGRP)* (`www.cisco.com/warp/public/103/1.html`), which is propriety to Cisco, a major router vendor. The advantage of distance vector routing protocols is their simplicity. RIP is easy to implement correctly, and the protocol works acceptably well on smaller networks. However, since the network nodes do not maintain a complete view of the network topology, there are limits to how much they can know about, and hence take advantage of, the available paths to a destination. In particular, the information available in RIP is so minimal that the protocol is unable to avoid slow convergence to correct routes when the internetwork is partitioned by failures.

Link state protocols are based on the idea that each router advertises the state of its links to other routers. As this information flows into a given router, it is used to create a map of the complete topology of the AS. This information is used to calculate complete routes and determine the correct next hop for moving a packet toward its destination. The most widely used link state protocol is Open Shortest Path First (OSPF) [42]. The fact that routers are provided with global

information is useful in determining good routes, but there is significant complexity involved in the protocol that propagates the link states. OSPF is one of the most complex of all RFC's.

BGP [48] is the dominant exterior routing protocol in the Internet. It is a hybrid between a distance vector and a link state protocol. On one hand, routers do not keep track of the global topology, but on the other hand, advertisements describe complete routes, rather than just hop counts. The selection of a best route to an AS is a function of both the local router's policies and the best routes as determined by its neighbors. That is, BGP allows distance metrics based on hop count to be overridden by policy-based metrics. This flexibility leads to potential short-comings which are discussed in Section 2.3 and throughout Chapter 6.

2.2 Routing in Ad Hoc Networks

A mobile, *ad hoc network* consists of a number of mobile nodes capable of communicating with their neighbors via broadcast or point-to-point links. In this setting, connectivity will be determined by signal strengths and the link technology which will be sensitive to noise and obstructions. Typical applications of ad hoc networks are disaster relief and battlefields, where a wired infrastructure is unavailable. Mobility of the nodes may cause connectivity to be extremely variable, so at any point in time, the nodes will be forming an ad hoc network with a different topology. In addition to that, the nature of the broadcast media typically makes the links low-bandwidth and lossy.

Nodes can be used as routers to forward packets. In this environment, the goal of a routing protocol is to discover and maintain routes between nodes. The essential requirements of such a protocol would be to:

1. React quickly to topology changes and discover new routes.
2. Maintain "good" routes so that data packet throughput is maximized.
3. Send a minimal amount of control information.

There is an obvious tradeoff between (1) and (2). Routing protocols like RIP and OSPF are assumed to be operating in a network with reliable links of a higher bandwidth, serving a stable family of hosts. This is why they do not satisfy (1) and they are not even particularly concerned about (3). On the other hand, such a network may not need complete connectivity between each pair of nodes at all times. In a disaster relief situation, it may be the case that only a few mobiles need to communicate, rather than every pair. Low bandwidth, unreliability, and rapidly changing connectivity can therefore be balanced against a potentially modest demand for end-to-end communication links by the use of *on-demand* routing. That is, routes can be discovered when

they are needed, thus potentially reducing the overhead of routing control messages.

Because of its simplicity, distance vector routing is a natural choice for routing in ad hoc networks. AODV (*Ad hoc on-demand distance vector*) protocol provides an instantiation of an on-demand form of distance vector routing that aims to keep control messages to a minimum. There are many other approaches to routing in ad hoc networks based on other strategies. These schemes are all grossly similar in complexity at the current time. AODV is more complex than RIP, not only because of the on-demand requirement, but also because of the presence of *sequence numbers*, which are the state added to the protocol to protect against loop formation. This will be the focus of our analysis of AODV.

2.3 Convergence of Routing Protocols

The process of establishing routes is called *convergence*. Propagation of routing information through the network (whether wired or ad hoc) takes time, so convergence does not happen instantaneously. Ideally, all the relevant information will eventually be distributed and the system will enter into a stable state where routes will stop changing. There are two important classes of questions that one can ask:

1. What happens with the routes before a stable state is reached? (i.e. Are they consistent? Are they loop-free? How persistent are looping routes?)

2. Are we guaranteed to reach a stable state at all? If yes, how soon?

The first question is important because incoming packets need to be routed *while* the protocol is converging. If a route is not valid during that time (i.e. it does not lead to the destination), a packet following it will be lost. A particularly bad case of invalidity is a looping route, where packets not only fail to reach their destination, but also waste bandwidth. Even if loops can form, we would like to know how long can they last and whether the protocol is guaranteed to break them at all. Loop freedom is an important issue for AODV, because links have a low bandwidth. On the other hand, stability is not a crucial issue for AODV, since topology is expected to constantly change and, consequently, the routes will soon become obsolete.

The situation is quite different in the case of RIP and BGP, where stability is important, because routes are likely to be valid for longer periods of time. In the case of RIP, a sharp real time bound for the convergence time also gives an upper bound for the life time of loops (which indeed can be created).

Convergence is an even more complicated issue for BGP. Varadhan, Govindan and Estrin [52] demonstrated circumstances in which routes to a given destination persistently oscillate. The extent to which it is a practical problem for BGP on the Internet is not well understood. Griffin and Wilfong [20] demonstrated that even if the BGP topology of the Internet were known, it would be infeasible in principle to decide whether it might display oscillations. Sufficient conditions for guaranteed convergence are presented in [19, 17]. Deeper understanding of the convergence properties of BGP is likely to be an active research area during the next few years.

Properties that a protocol is supposed to satisfy are called *requirements*. Routing protocol requirements are often specified during the design phase. Proving or disproving them early can steer the design efforts in the right direction. Requirements can also be established for an already existing protocol. Showing that a protocol satisfies or breaks them can help us to better understand the protocol, use it more efficiently and quickly detect origins of errors.

Network protocols are often analyzed through performance-based testing. While such tests provide invaluable feedback, there are several reasons why they alone do not suffice for validating a protocol design. Testing informs us about the average behavior of the protocol, while, in order to establish satisfaction of the requirements, one needs to study the worst-case behavior. Furthermore, testing is usually bound to a specific protocol implementation which may not demonstrate some more general errors in the protocol design. Finally, there are some protocol properties which are not directly related to performance. Many security properties, for instance, fall into this category. This is why performance-based testing should be supplemented by formal correctness analysis.

2.4 Summary

We have introduced the basics of the routing infrastructures in IP and ad hoc networks. IP routing protocols can be *internal* and *external*. An internal protocol routes inside a single administrative domains, while an external protocol routes between administrative domains. Examples of internal protocols are RIP and OSPF. BGP is the principal external routing protocol. Internal protocols can be further divided into *distance vector* and *link state* protocols.

In ad hoc networks, nodes are mobile and communicate via wireless links. Because of that, the network topology constantly changes and routing becomes more challenging.

Convergence is the process of route discovery. We are interested in the properties of routes produced during convergence and the length of convergence. Not all routing protocols are guaranteed to converge.

Chapter 3

Methodology

3.1 Verification Attributes of Routing Protocols

There has been a variety of successful formal studies of communication protocols. However, most of the studies to date have focused on *endpoint* protocols (that is, protocols between pairs of hosts) using models that involve two or three processes (representing the endpoints, or the endpoints and an adversary, for instance). Studies of routing protocols must have a different flavor since a proof that works for two or three routers is not interesting unless it can be generalized. Routing protocols generally have the following attributes which influence the way formal verification techniques can be applied:

1. An (essentially) unbounded number of replicated, simple processes execute concurrently.
2. Connectivity is dynamic.
3. Processes are reactive systems with a discrete interface of modest complexity.
4. Real time is important; many actions have real-time constraints or are carried out in response to a timeout.

Most routing protocols have other attributes such as latencies of information flow (limiting, for example, the feasibility of a global concept of time) and the need to protect network resources. These attributes sometimes make the protocols more complex. For instance, the asynchronous version of the Bellman-Ford protocol is much harder to prove correct than the synchronous version [5]. The RIP standard, which is based on the Bellman-Ford protocol, is still harder to prove correct because of the addition of complicating optimizations intended to reduce latencies.

The fact that our system consists of a set of *replicated* processes can greatly simplify the verification. This is due to a large number of symmetries that can be found in such a system. For instance, every property that claims something about *every router* in the system can be proved by demonstrating it on an arbitrarily chosen *single router*. If routers were allowed to behave differently (i.e. run different protocols or have different roles in the same protocol), this kind of reasoning would be much more complex. For the same reason, a router's *environment* which consists of a homogeneous collection of identical routers can be represented relatively easily. We discuss this in a greater detail in Chapters 4 and 5. An important distinction between modeling environments in routing protocols and often studied security protocols is the fact that security protocols require a model of the adversary. This can be a controversial point, since the capabilities of a potential adversary are not known in advance. In the case of routing protocols, the network environment plays the closest role to the adversary. Although we do need to model the network attributes such as reliability and dynamics of the topology, the assumptions here are much clearer.

We view routing processes as event-driven reactive systems. An event can generally fall into one of the following three categories:

1. An external event generated by the environment (e.g. a link breakage or restart of the routing process).
2. Receipt of a packet.
3. A local timeout, indicating that a certain amount of time has passed since a timer was last set.

When an event happens, routers can perform actions. Examples of common actions include routing table updates, sending of packets, and setting up timers.

3.2 Formal Methods

Parts of the work described in thesis are greatly aided with the use of automated reasoning tools. This section gives a broad overview of the currently existing technologies in this area, as well as specific techniques that we used.

The development of general-purpose tools for automated reasoning is motivated by several important factors: complexity of today's systems, increasing error costs, and commonality in reasoning frameworks. Let us briefly look at each one of them separately.

The efficiency, flexibility, and low cost of computer technology have increased the demand for automated systems which are supposed to handle increasingly complex tasks. Consequently, the

system architectures have become increasingly complex as well. These architectures are well past the point where their designers can be reasonably confident about the system correctness. Moreover, complicated tasks make it hard to even precisely state what constitutes the system correctness (i.e. system requirements).

The widespread deployment of computer technology, especially in safety-critical systems, makes the design errors very costly. There are many situations in which valuable material assets or even human lives are at stake. The same software or hardware error which was relatively benign a couple of years ago may have a huge cost today, because the system is used in a different environment.

Finally, formal reasoning about different systems is often not entirely different. There are many common elements, which include mathematical models, logics, reasoning techniques, abstraction mechanisms, and others. Therefore, a single general-purpose tool which embodies these mechanisms can aid the formal reasoning about many different systems. This reduces the overhead cost of using such tools.

The factors discussed above demonstrate a clear need for systematic procedures for assuring that a system does what is supposed to do. The term *formal methods* generally refers to the collection of techniques that are based on rigorous mathematical reasoning and are designed to aid the process of discovering system errors. In this case, a "system" can be software, hardware or any other system whose behavior can be accurately described in formal language. For example, formal methods can be used to reason about correctness of an ambulance vehicle dispatching protocol. This is not necessarily a computer system (although it may be), but it presumably has precise rules which can be formally analyzed. The analysis may, for instance, attempt to verify that the protocol satisfies the following property: *if the frequency of emergency calls is less than f, then each call is assigned to a vehicle which is initially less than 5 miles away.* Software and hardware still form the largest part of the application domain for formal methods due to the factors discussed earlier in this section.

The techniques presented in this document use formal methods mainly for *verification*. Verification can be thought of as a "top-down" process in which an already designed system is checked for satisfaction of certain requirement properties. For example, we may want to verify that our telephone system satisfies the following invariant: *if a phone is on-hook, it is not connected.* A dual of verification is *synthesis*. Synthesis is a "bottom-up" process which starts from a set of requirements and builds a system that satisfies them. An example would be a problem of synthesizing a program that computes a function implicitly described with some property (e.g. it is linear in the first argument and quadratic in the second).

We see that for both verification and synthesis, it is important to know the *requirements*—the set

of properties that the system is supposed to satisfy. Coming up with the right set of requirements may sometimes be harder than building a system that satisfies them. An illustrative case study can be found in [7]. In the case of verification, we must also have a formal model of the system, which is called the *system specification*. A more detailed discussion of requirements and specification is presented in [22].

Given a specification and a set of requirements, the problem of determining whether the specification satisfies the requirements is called the *verification problem*. Typically, a specification contains description of the system state and rules that specify how the state changes. Specifications are often *nondeterministic*. This means that the system can perform several different actions at a given time. This is because the actions may depend on some external factors that become known only during the execution. Such external factors include user inputs, order and timing of received messages, random numbers, and others. Verification of nondeterministic specification requires reasoning by cases. A case analysis can be deeply nested, which makes the task of keeping track of the cases quite challenging for traditional paper-and-pencil proofs. A machine with an appropriately chosen reasoning tool can provide significant help in those situations.

There is a variety of tools that can be used for this purpose. The two most popular kinds of verification tools are *model checkers* and *theorem provers*. The techniques they use are quite different, showing a tradeoff between expressibility and automation. This division is not strict—there is a range of tools that combine the features of both approaches.

Model checkers are completely automatic tools. A user needs to provide a model (i.e. a specification) and a requirement property. The model checker performs the rest of the verification. Upon completion, it returns one of the two answers:

- A "yes" answer, indicating that the specification satisfies the requirement.
- A "no" answer, together with a scenario that demonstrates violation of the requirement by the specification.

A model contains variables whose values change according to some rules. Every combination of the values of all the variables defines a *system state*. The system state may also include contents of the communication channels, set of active processes, a program counter for each processes, etc. The set of all states reachable by a given system is called the *state space*. Model checkers perform verification by exhaustively exploring the state space while looking for requirement violations. The size of the state space can significantly affect the verification feasibility. Verification of systems with large state spaces can quickly consume all the available memory and run for a prohibitively long time. This happens quite often in practice, since state spaces suffer from the problem of

combinatorial explosion—the size of the state space grows exponentially with the size of the model. Consequently, model checking techniques do not scale well, which is arguably considered to be their biggest impediment. Model checkers can generally perform only verifications on finite state spaces. In some cases, infinite state space verification can be carried out by using finitary abstractions. Some model checkers can automatically come up with such abstractions. We demonstrate examples of manually constructed finitary abstractions in Chapter 4. Another technique designed to cope with the problem of large state spaces is *symbolic model checking*. Instead of searching through individual system states, a symbolic model checker searches through *regions* of states. Regions correspond to (and are represented as) logical formulae [30]. Model checkers have been successfully applied in different industries, with hardware design being one the most popular application domains. Widely known model checkers include COSPAN, SPIN [27, 51], Mocha [3] and SMV [50].

Contrary to model checkers, **theorem provers** usually do not have such a high degree of automation. Correctness statements are formulated as theorems in the logic of the theorem prover. A theorem prover can do parts of the proof automatically, but it often requires "hints" from the user which are used to steer through the proof-search process. This is inevitable, since most of the nontrivial logics are undecidable. However, expressibility is a significant advantage of theorem provers. One can express and prove much more general results than with a model checker. In principle, any piece of general mathematics (e.g. any theorem found in a mathematics textbook) can be formalized by a general-purpose theorem prover. Unlike in model checkers, size of the state space is not a big concern here. In fact, theorem provers are so general that they are often used to reason about infinite-state systems. In practice, though, some proofs may be difficult to carry out. This greatly depends on the user's skills, built-in support for automation, and the set of available previously constructed theories that can be used as a foundation. Theorems can usually be constructed in a *top-down* or a *bottom-up* fashion. In a top-down proof, the user starts with a desired theorem as a *goal*. At each step, a *tactic* is applied to the current goal, which transforms it into a hopefully simpler goal or proves it completely. Tactics can perform induction, instantiate existentially quantified variables, split conjunctions, disjunctions and implications, etc. Some theorem provers even allow users to define new tactics, so one can build special-purpose tactic libraries. This is an extremely useful feature of HOL [25, 18]. In a bottom-up approach, new theorems are build from the already proved ones by using generative rules (e.g. modus ponens, generalization, conjunction). Besides HOL [25, 18], some of the more popular theorem provers are PVS [46], Coq [11], and ACL2 [2].

3.2.1 Applications of Formal Methods

As mentioned before, formal methods have been successfully used for hardware verification. Hardware designers create extremely complex digital circuits. It is important to make sure that a circuit really has the intended functionality under all possible circumstances. A 64-bit circuit may have several billions of billions of states. Clearly, correctness analysis of such a circuit exceeds the boundaries of even the most ambitious human efforts. A model checker may be the right tool for this job. There are two principal reasons for that:

- Digital circuits have precise requirements and design description.
- Verification usually boils down to a huge case analysis, where large groups of cases are handled in a similar routine fashion.

In some situations, a state space may be so large that even a model checker may not be up for the task. One way to cope with this problem is to use *modular design*. A modularly designed circuit, for instance, consists of a set of sub-circuits (modules), each with its own description and requirements. Each module is model-checked against the set of its requirements. Modules are composed in such a way that their requirements together entail the global requirement—this fact usually needs to be formally verified too. Discussions on modular and "divide-and-conquer" approaches can be found in in [1, 4, 24]. Modular design can greatly reduce the required verification effort. In some cases, a circuit consists of a large number of identical sub-circuits, whose correctness then clearly needs to be verified only once. Modular design is a general technique which is also applicable outside the hardware domain. Some model checkers [3] explicitly support modularity as the main design principle.

Despite of the state space reduction techniques such as abstractions and modular design, we may end up with a system complex enough to make a complete verification infeasible. In such cases we can use *testing*. Testing usually does not assure that the system is completely correct, but it can reveal bugs. For instance, some model checkers can perform randomized and/or user-guided simulations of the system. Simulations produce *traces* of the system behavior which can then be compared against the requirements. Clearly, testing is not guaranteed to find all errors. However, a carefully designed test suite can guarantee a large coverage of the possible scenarios.

Testing usually reveals errors quickly in the beginning and slower later on, since errors are being fixed. Eventually, when no more errors can be discovered, one may try to prove complete correctness of the system with a theorem prover. Attempting to prove correctness earlier may be inefficient, because theorem proving typically reveals errors at a much slower rate. Moreover, if an

error is discovered deep in the proof, the entire proof needs to be redone once the error is fixed. Nevertheless, errors discovered by a theorem prover are typically more informative. While a model checker outputs a single error scenario, a theorem prover will usually provide a logical description of the circumstances where the error occurs. Such description will be given as a set of assumptions which, unlike in true theorems, does *not* imply the theorem statement. This can reveal subtle boundary-case errors, which occur so rarely that they are hard to catch by randomized testing.

One can imagine the following verification strategy based on the techniques described above:

1. Try to model the system and and verify it completely using a model checker.

2. If the state space is too big, use abstractions and modularity.

3. If the state space is still too big, use testing and eliminate all the discovered errors.

4. Finally, try to prove correctness by using a theorem prover.

A variety of special-purpose verification techniques have been developed for the cases where general-purpose techniques do not provide adequate support. Below we identify several active areas of research which try to develop such specialized techniques.

Program verification focuses on the problem of proving that a given program (whose semantics is well defined) satisfies a given property. Programs usually have large or even infinite state spaces. A variety of language-specific techniques have been developed to cope with this problem. Some of them are predicate abstractions, control flow analysis, and syntactic and semantic program transformations.

Proof carrying code is an interesting application of program verification. It is based on the assumption that software users will often want to run programs written by non-trusted parties and still have the confidence that the programs will not do anything “bad”. A solution is to pack the program together with a proof of the program correctness. A user can check the proof using *any* suitable proof checker. The proof checker can be designed by a third party or perhaps by the user herself. This reduces the chance of a bogus proof being accepted, since that would require the proof and the checker to be corrupted in a consistent way. Some compilers can automatically generate proofs about programs during the compilation. This method can be useful even when constructing a proof is hard and not automated, since proofs are likely to be constructed once and then used many times.

Type theory can also be viewed as a class of formal methods. It, too, investigates techniques for providing guarantees about program behavior, but from a more practical angle. There are several distinctive features of types as formal methods. Firstly, the logic of program properties is

very restricted and often decidable. It is embedded in a type system which is usually presented as a set of Plotkin-style rules. The intention is to have an efficiently decidable logic which can then be completely automated and embedded in the compiler. Secondly, proofs are not explicitly exported. They only need to be *generated* by the compiler. The compiler is fully trusted in this case—if it fails to generate a proof, an error is reported and the executable will not be generated. Thirdly, a user has little or no freedom in choosing the theorems to be proved. The choice is automatic, directed by the type system, which presumably captures the essence of a "good" program behavior. This is why type systems, unlike logics in general, are designed with a particular programming language in mind. Their main task is to catch low-level errors and indicate their origin with some reasonable accuracy.

Protocol verification is an area of a great practical interest. It includes verification of security protocols [41, 29], communication protocols [49, 12], cache coherence protocols [14, 13], mutual exclusion protocols [47], and others. Unlike program verification, protocol verification always deals with multi-party systems that are usually *simple* and *distributed*. Protocols are popular targets for formal verification for several reasons:

- They have precise and relatively simple descriptions, which makes modeling easy.
- Their correctness is important, because they are used frequently.
- They often have small state spaces.

One exception from the last point are n-party protocols. These are the protocols which do not have a bound on the number of parties involved. Consequently, their state space is unbounded too. This makes it impossible to carry out a verification by performing a straightforward state space exploration. This is one of the reasons why there has been so little work on the automated analysis of n-party protocols, relative to other kinds of protocols. Many networking protocols (including routing protocols) are n-party protocols.

3.3 Verification Methodology

This section describes our five-stage verification methodology. The stages of analysis are: standard, pseudo-code, specification, requirements and verification. We will discuss each of them at a greater level of detail. When discussing tradeoffs involved at each stage, we focus on the issues related to the use of automated support, rather than manual proofs. Manual verification gives us the unlimited freedom in choosing formalisms, so many of the tradeoffs become irrelevant or, at least, a matter of individual taste.

3.3.1 Standard

Routing protocols are supposed to run on routers which are designed by different manufacturers. Because of the very nature of routing, it is essential that all the different routers work together and understand each other. This interoperability requirement is the main motivation for the routing protocol *standards*. Standards are documents that precisely describe how protocols work. They are given in the form of IETF [1] RFCs [2] or standard drafts. Routing protocol standards have the following parts:

- Interface description (packet types and formats).
- Algorithms (how to process incoming data, how to maintain state).
- Intended usage (when to use the protocol, how to configure it).

The idea behind a standard is to give enough guidance for each manufacturer do design its own implementation which will be compatible with any other implementation designed in the same way. Even though implementation details may greatly vary from one manufacturer to another, the standard itself should guarantee that different implementations would work together.

Finding Errors vs. Proving Correctness. Software engineers often recognize the following three artifacts as essential for reasoning about software correctness: the *requirements*, the *standard* and the *implementation* [3]. The desired state of affairs is the standard which refines the requirements and the implementation which refines the standard. This thesis focuses on the former refinement relationship—verification of routing protocol *standards* with respect to the requirements. Establishing this relationship is important, because it reduces the problem of verifying correctness of an implementation to verifying its adherence to the standard. Moreover, the standard needs to be verified only *once* and the result can then be used for verification of *many* implementations.

In practice, however, the implementor often faces a situation where correctness of the standard is not formally established. In that case, one can try to formally verify the implementation from scratch. This is often too expensive, especially if the implementation is likely to change in the near future. Another, more practical solution is testing. The implementation can be used to produce execution traces, which can then be formally analyzed. In [8] we have described one such architecture for analyzing routing protocol implementations by using a network simulator. If an implementation trace demonstrates an error, it is important to know whether that error comes

[1] Internet Engineering Task Force
[2] Request for Comments
[3] A more detailed study of all the artifacts is presented in [22].

from the implementation (i.e. incorrectly implemented standard) or from the standard (i.e. the standard violates the requirements). A framework for determining which of the two cases applies is presented in [9].

Both verification and error-searching can help increase the confidence in software correctness. They are different techniques which should be applied under different circumstances. In the rest of this thesis, we will generally focus only on verification issues.

3.3.2 Pseudo-code

In order to analyze protocol standards, we first need to translate them from plain English into some formal framework. In addition to enabling formal analysis, this process can also identify ambiguities in the standard. Such ambiguities are often sources of errors.

The first step towards formalization of a protocol standard is describing it in pseudo-code. The main goal of this phase is to achieve a good tradeoff between preciseness and readability of a protocol description. For this purpose, we have designed a special-purpose pseudo-language called POPSCL [4] (**P**rotocol **O**bject **PS**eudo-**C**ode **L**anguage). The language is an extension of the specification style used by Carolyn Talcott and J.J. Garcia-Luna-Aceves. POPSCL code is divided into six sections:

Constants section lists some fixed or locally configured constants that the routing process uses.

State describes a router's state. This includes variables, tables, and timers.

Initially describes the initial state of a router.

Events lists the events that the routing process recognizes.

Event handlers describes what actions are carried out as a response to an event (e.g. setting timers, updating tables).

Utility functions describes functions that the routing process can invoke; these may generate events recognized by other routing processes (e.g. broadcasting a packet to all neighbors).

The name of the language is inspired by its resemblance to the object oriented style. A process description resembles a class, which contains variable declarations (*State*), method declarations (*Events*), method definitions (*Event handlers*), and a constructor (*Initially*). A class may also have available some external functions (*Utility functions*).

[4] Pronounced as *popsicle*.

POPSCL supports a special kind of variables called *timers*. Timers can be thought of as "stopwatches". They continuously decrease their value as long as it is greater than zero. When a timer reaches zero, it generates a *timeout* event. Just like a stopwatch, one can **set** a timer to a specific value, or **deactivate** it. The current value of a timer (i.e. remaining time before the timeout) can be read at any moment.

Our syntax for any kind of "packet send" operation requires that contents of the packet be enclosed in rectangular brackets. Our packet format generally reflects the logical, rather than the physical structure. In some cases, a protocol (e.g. AODV) needs to use the IP destination field of an IP packet. We include that field at the end, separated by a semicolon from the logical contents. A typical packet is hence denoted as $[logical\ contents; IP_DEST]$.

3.3.3 Specification

POPSCL is a pseudo-code language and as such, it does not have a precise semantics. In particular, details about the network and external functions are not specified. In order to fully formalize the standard, we now need to translate it from POPSCL into a formal specification language. Such languages have a precise semantics which can be used to reason about the specification. Different languages offer a range of tradeoffs regarding the level of abstraction, expressibility and elegance of the specification. We regard the following aspects as being the most relevant for specifying routing protocols:

Processes: The first requirement of a specification language is its ability to specify behavior of a single routing process. We should be able to describe its state and its reaction to events.

Communication: We should be able to describe message passing between protocol processes. For routing protocols, messages are encapsulated in packets. Message passing may be synchronous or asynchronous, and may have unicast or broadcast semantics.

Packet Formats: Because packets are essential for process communication, we need to have support for describing packet formats. While the details about a packet's physical layout may not be interesting for formal analysis, its logical contents will be important. It is desirable to have types for packet fields. This makes it easier to handle the field values. In addition to that, a type checker can automatically detect modeling errors such as incorrect packet formats.

Network Attributes: Routing protocols run over complex multi-node networks. We need to be able to describe static and dynamic network topologies. Other attributes that need to be

modeled include reliability (*can packets be dropped or reordered?*), functionality (*broadcast vs. unicast vs. neighborcast*) and the traffic model (*when and where are packets being sent?*).

Timers: All routing protocols have real-time specifications that are implemented with timers. They are visible in our POPSCL descriptions. Although timers are generally important, they can be irrelevant for some properties (e.g. certain invariants). Because of that, the importance of the support for timers inside a specification language will depend on the properties that we ultimately want to verify.

We have experimented with two principally different specification languages. We discuss and compare their features below.

Promela. Promela (PROtocol MEta LAnguage) is a specification language of the model checker SPIN. As its name suggests, the language is designed for specifying communication protocols. It has a natural notion of a process. This makes it a suitable language for specifying systems containing a fixed number of processes.

Promela data types include basic types, arrays and struct declarations, which are generally expressive enough for our purposes. For instance, struct declarations are typically used for describing packet formats and arrays are used for describing the topology.

There are built-in primitives for synchronous and asynchronous channel communication between processes.

Promela has an *interleaving execution model* [26, 27, 51]. This means that at every step, only one process executes an instruction [5]. Scheduling is completely nondeterministic, which means that process activations can be interleaved in an arbitrary way.

Network topology needs to be hard-coded in Promela. We represent the topology using an adjacency matrix and a *network process* which captures all messages on the output channel of a node and sends it to the input channel of the appropriate neighbor. In the case of a broadcast, the message is sent to all the neighbors. We can simulate a dynamic topology to a certain extent, by bringing links up and down. However, the number of nodes needs to be bounded, since the total number of states must be bounded. In Promela, it is difficult to specify a class of topologies defined by some high-level property. For instance, it is hard to nondeterministically choose a *connected* topology (e.g. a topology where there are paths between all pairs of nodes).

Modeling simple reliability assumptions in Promela was straightforward. Examples of such assumptions are "*packets can be dropped arbitrarily*" or "*packets can never be dropped*". However, reliability assumptions of the kind "*at most k out of every n consecutive packets get dropped*" are

[5] An exception is synchronous communication, where a *send* and a *receive* execute simultaneously.

a bit trickier to model. We have encountered assumptions of this kind related to AODV. Their modeling requires additional state, which increases the state space and slows down the verification. Besides the network topology, this is another example of Promela's all-or-nothing nondeterminism, which is not quite sufficient for fine-grained environment descriptions.

Another weakness of Promela is the absence of support for timers. We simulate the effect of timers by making them completely nondeterministic. In other words, we assume that actions can take arbitrary time to complete, so that timeouts can happen at *any* time. This can be done only in situations when the property to be verified does not depend on the exact values of timers. In such cases, timers simply restrict the set of possible *orderings* of the events, but not their *timing*. By modeling them nondeterministically, we extend the set of possible runs of the system. When performing verification with SPIN, the claim is of the form "*in every run of the system, some property holds*". Therefore, it is technically possible that we detect a violation which can not be realized in the real system. However, if no violation is detected, then the original system does not have violations either. In other words, we can have false negatives, but not false positives verification results. Having said that, we should point out that we have never encountered a false negative answer in our verification. The reason is the fact that most of our properties were not sensitive to timer values. It was often enough to only assume that a timer will *eventually* generate a timeout (at some unspecified moment), to be able to verify the property.

Higher Order Logic (HOL). Both HOL and PVS are theorem provers for the classical higher order logic based on Church's simple theory of types. Higher order logic is the specification language of both HOL and PVS, although the type systems of the two slightly differ. It is a very expressive language and can be used to specify every aspect of a routing protocol precisely. However, we will concentrate on aspects that can be "naturally" expressed in HOL in a way that will facilitate proofs of modest complexity later on.

HOL does not have any inbuilt support for modeling processes. Instead, processes need to be encoded as state transformer functions. Because of that, HOL specifications are fairly high level and can not capture process dynamics at the same level of granularity as Promela. For example, state changes in HOL occur instantaneously, whereas in Promela, each instruction can change part of the state. As a result, HOL specs do not allow analysis of some low-level aspects of process execution, such as deadlocks. Asynchronous communication with latency is hard to model in the HOL setting. Instead, we model communication as an instantaneous ("handshake") event between a sender and a receiver. Multicast is modeled as a sequence of interleaved communications, one for each receiver. Packet formats are encoded using HOL record types which have roughly the same

expressibility as Promela structs.

HOL can easily express any class of topologies with n nodes and arbitrary dynamics. A single topology is represented as a graph, and specific properties of topologies are expressed as HOL predicates over graphs. Examples of such predicates include connected graphs, bipartite graphs and graphs of a given diameter.

There is no inherent support for timers in HOL and they are encoded in a similar fashion as in Promela.

Because of the high level of abstraction, HOL can not capture very detailed scenarios. For example, since communications are viewed as instantaneous, we can not model reordering of messages. On the other hand, HOL is fairly convenient for describing fairness constraints, such as "*no router ever stops communicating with its neighbors*".

3.3.4 Requirements

In order to formally establish correctness of a routing protocol, we need to be able to to formally express the requirements. The specification and requirements languages should have similar semantic models. It is often the case that the language of requirements is a simpler and more abstract version of the specification language. We discuss the tradeoffs between LTL, Promela, and HOL as requirements languages for routing protocols.

LTL (Linear Temporal Logic). It is used in conjunction with Promela in SPIN. LTL describes properties of infinite sequences of states (i.e. *traces*).

Figure 3.1: LTL grammar

$$\varphi ::= p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \Box\varphi \mid \Diamond\varphi.$$

Figure 3.1 shows the grammar of LTL. There are two temporal operators: $\Diamond$ ("eventually") and $\Box$ ("always"). p is a proposition—a predicate that can be evaluated in a single state. LTL is a fairly common modal logic for expressing properties of reactive systems. Detailed syntax and semantics can be found in [40]. LTL is simple and very useful for expressing properties about one or some fixed number of nodes. However, it can not express properties about an arbitrary number of nodes. An example is the *loop freedom* property. Each router maintains a `next` pointer for each destination. This pointer specifies which neighbor will be used as the "next hop" to forward packets designated for that destination. The loop freedom property states that `next` pointers do not form

cycles. This property can not be expressed in LTL directly, unless we know the exact number of routers in the system. Even in that case, the formula would most likely be large and cumbersome. A trick which we use to express properties like loop freedom in LTL is to reduce them to some *local* invariant. For example, in order to establish loop freedom, it suffices to find a strict ordering $\prec$ on the set of nodes with the following property:

$$\forall n_1, n_2.\ (\mathsf{next}(n_1) = n_2) \ \Rightarrow\ (n_1 \prec n_2).$$

Notice that this property depends only on two nodes and therefore can be expressed in LTL.

Promela. SPIN translates LTL formulae into Promela and Promela programs into Extended Finite State Machines (EFSMs) [26], which are ultimately used in verification. Verification is reduced to the language containment problem on the automata representing the specification and the requirements. The user can encode the requirements directly in Promela. If a property can be expressed both in Promela and LTL, it is usually simpler to express it in LTL. However, there are cases when it is more convenient to use Promela. Promela is strictly more expressive than LTL. It provides, for instance, a more precise access to the variables (i.e. we can capture their value at any point). This is typically useful when we need to remember the state at some point in the trace and use it at some later point. Another example is counting, which is hard to do in LTL. A sample property that uses counting would be "*after 5 packets are received, the value of x is negative*".

Nevertheless, situations where such fine-tuned control is needed for the requirements are relatively rare. Routing protocol requirements generally depend solely on the state and not on the event history. That is why things like counting are often unnecessary in this context. Requirements are generally simpler than the specification, and the similar relationship holds between their corresponding languages. Given that LTL is simpler than therefore less prone to human errors, our general recommendation is that Promela should not be used to state requirements that are within the expressibility range of LTL.

HOL. Provides the most expressive description technique, since both LTL and Promela can, in theory, be embedded in Higher Order Logic. We pointed out before that HOL, with its declarative, logical nature, is not a perfect ambient for modeling processes. However, the situation is quite the opposite with requirements, which are naturally declarative—they state *what* is supposed to be true, not *how* to make it true. HOL requirements suffer from other problems inherited from the HOL specifications—namely, they are fairly high-level, which makes it difficult to express some communication and concurrency properties (e.g. deadlock freedom). However, most state properties (such as convergence or loop-freedom of routes) are easily expressed in HOL.

3.3.5 Verification

We have used two automated verification techniques: theorem proving and model checking. The largest part of our theorem proving work was done in HOL. We re-did a part of our RIP verification in PVS, so that we can compare the tools. Model checking was performed using SPIN. Below we summarize advantages and disadvantages of both techniques.

Model checking in SPIN. SPIN takes as inputs a model description in Promela and a requirement property (in LTL or Promela). It then performs an exhaustive search on the model's state space while searching for violations of the requirement. If SPIN finds a run that violates the requirement, it aborts the verification and presents the counterexample. Otherwise it informs the user that the model satisfies the requirement. Since Promela can only specify systems with a fixed number of processes, the same limitation carries on to SPIN verification. Therefore, SPIN can not be directly used for *full verification* of routing protocols. However, SPIN can perform *instance verification.* By instance verification, we mean verification of a protocol in a specific scenario (i.e. fixed topology and traffic pattern). Instance verification has two main purposes:

1. It quickly detects modeling errors. Instance models can be simulated in SPIN and the user can often quickly detect basic errors, such as incorrect computation of optimal routes and unexpected deadlocks.

2. It can quickly detect requirements that are not met by the specification. In most cases, a violable requirement has a simple counterexample, containing only two or three nodes.

In addition to that, full verification can sometimes be reduced to an instance verification. Theorems of the following kind can justify such reductions: *"If an error exists in the general n-routers case, then an error can be demonstrated in a 3-router configuration."* We call such theorems *instance reductions.* Instance reductions are typically proved by using a theorem prover. When such a reduction is proved, we only need to verify, say, 3-router configurations (using SPIN, for instance) in order to establish the full correctness. However, even instance verification can be too burdensome for SPIN, since the state space generally grows exponentially with the number of nodes. In some cases, this prevents SPIN from performing even small instance verifications, with three or four nodes. In such cases we need to use *abstractions.* Abstractions are similar to instance reductions in the sense that they reduce an unbounded verification to a finite verification. However, in the case of abstractions, the reduced system does not need to be an instance of the protocol. Moreover, even the property that is to be verified can change. Formally, we say that a pair (M^{abs}, P^{abs}) of a

model and a property is an abstraction of the pair (M, P) if the following holds:

$$M^{abs} \models P^{abs} \implies M \models P.$$

The above implication does not have to be the equivalence. This potentially sacrifices soundness of the verification, but preserves completeness. We only require that correctness of the abstraction implies correctness of the original system. The other direction is useful when the verification of the abstraction fails. In that case we can not conclude anything about the original system. The original system may still be correct, but our abstraction is too "strong". Equivalence, on the other hand, would allow us to conclude that the original system is also incorrect in that case. We see that instance reductions form a special class of abstractions. Notice that the equivalence always holds for instance reductions.

In conclusion, we can say that the main advantage of verification through model checking is its full automation. It is limited by the expressive power and the exponential growth of the state space, but whenever model checking is possible, it is usually the most efficient way of verifying a conjecture.

Theorem proving in HOL/PVS. In order to prove that the model satisfies the requirements in HOL, one needs to formulate a conjecture and prove it, hence deriving a theorem. A theorem (or a conjecture) in HOL is of the form $A \vdash t$, where A is a list of formulae that represent the assumptions and t is a single formula that is the statement of the theorem. The meta-language of HOL is ML, which conveniently uses an abstract datatype `theorem` to represent theorems. Objects of the type `theorem` can only be created in a controlled way—by using special functions. This approach reduces proof checking to type checking, so that theorems are considered proved solely because the ML type checker has derived the type `theorem` for them. There are two ways of deriving a theorem in HOL: bottom-up (or forward) and top-down (or goal-oriented).

In a bottom-up way, a theorem is derived from other theorems through inference rules. For instance, given the theorems $A_1 \vdash t_1 \vee t_2$ and $A_2 \vdash \neg t_1$, our inference rule may produce the theorem $A_1 \cup A_2 \vdash t_2$. This is sound, because the existence of the first two theorems implies the existence of the third one. This rule can be represented as an ML function of the type `theorem->theorem->theorem`. The function would generate an exception if t_1 from the first two theorems do not match.

In a top-down way, a conjecture is stated as a *goal* that is to be proved. At each step, the user applies a *tactic* to a goal, which transforms it to a hopefully simpler goal. A tactic is a function which takes a goal as an argument and returns another goal (or a list of subgoals) and a justification of the reduction. The justification indicates how to prove the original goal, given the proof(s) of the returned goal(s). For instance, a tactic may split the goal $A \vdash t_1 \wedge t_2$ into the subgoals $A \vdash t_1$

and $A \vdash t_2$. It is easy to see that, if the subgoals are proved, the original goal can be proved as well.

Tactics may have arguments. For instance, applying the existential-quantifier-elimination tactic `(EXISTS_TAC 9)` to the goal $A \vdash \exists x.x > 2$ reduces it to the goal $A \vdash 9 > 2$. Tactics are also represented by a separate ML datatype. The fact that users can develop their own new customized tactics in ML is one of the greatest strengths of HOL.

Despite of the powerful logical machinery, proving even simple theorems about routing protocols can be hard. The main reason is the low degree of automation. The user often needs to spend a significant portion of time on proving trivial lemmas. The payoff is a high confidence in correctness of the results.

The situation is somewhat different with PVS. PVS has powerful decision procedures which can automatically prove many conjectures that would require step-by-step proofs in HOL. These procedures are based on various heuristics and are implemented with fast algorithms. While this kind of automation can save a lot of time, it is also less rigorous. Unlike HOL, PVS can not produce *proofs* which would be checkable by an independent proof checker. This is because parts of the proof are hidden in the algorithms used by the decision procedures. In some sense, PVS behaves like a model checker for higher order logic.

To summarize, theorem proving has the following major roles in verification of routing protocols:

1. Proving the soundness of the abstractions.

2. "Gluing" the results obtained from model checking with other theorems.

3. Full verification, in cases when it can not be done with more automated tools.

Theorem proving can also be used for instance verification, especially in situations when a model checker can not cope with the large state space. The advantage of theorem proving in such cases is the fact that, unlike in model checking, the needed verification effort usually does not increase with the size of the instance. Moreover, it is often easier to prove a theorem for a general n-router configuration, than for a configuration with a specific number of routers. This is not to say that instance verification is easily done by theorem proving. It does require significant assistance from a human in proof searching. In return, however, it provides a proof that can probably be generalized to a full verification.

3.4 Summary

Routing protocols have a number of common verification attributes. *Formal methods* are techniques based on rigorous mathematical reasoning whose purpose is discovery of design errors. The most common formal method tools are *model checkers* and *theorem provers*. Theorem provers are less automated and more expressive, while model checkers are fully automated and less expressive. Our verification methodology has five artifacts: *standard*, *pseudo-code*, *specification*, *requirements* and *verification*. Standard describes the protocol design in plain English. Our pseudo-language POPSCL is designed to concisely describe routing protocol standards in the form of pseudo-code. Specification is a fully formalized description of the protocol. Requirements are the properties that are supposed to be satisfied by the protocol. Verification is the process of determining whether the specification satisfies the requirements.

Chapter 4

Routing Information Protocol (RIP)

This chapter describes our first case study—verification of RIP. RIP is a classic example of a distance vector routing protocol. We derive two main results: proof of convergence and a sharp real-time bound on convergence time.

4.1 Protocol Description

RIP protocol works inside a single autonomous system, such as the one depicted on Figure 2.1. The main task of a router is *forwarding*. When a router receives a packet, it first looks at the IP address of the destination network of the packet. Based on that address, the router needs to forward the packet to the next router by sending it out along one of the interfaces. The next router will perform a similar decision process; this repeats until the packet reaches its destination. The overall goal is to minimize the number of "hops" that a packet traverses on its way to the destination. The principal task of RIP is to instruct the routers on how to forward packets to achieve this goal.

Each RIP router maintains a routing table. Routing table contains one entry per destination network, representing the current best route to the destination. An entry corresponding to destination d has the following fields:

- hops: number of hops to d (i.e. total number of routers that a message sent along that route traverses before reaching the network d - this includes the router where this entry resides). This is called a *metric* for d.

- nextR: next router along the route to d.
- nextIface: the interface which will be used to forward packets addressed to d. It uniquely identifies the next network along the route.

Routers periodically advertise their routing tables to the neighbors. An advertisement is a pair (d, hops) of a destination network address and a metric for that destination. Upon receiving an advertisement, the router checks whether any of the advertised routes can be used to improve current routes. Whenever this is the case, the router updates its current route to go through the advertising neighbor. Routes are compared exclusively by their length, measured in the number of hops.

For instance, suppose that a router r currently has hops $= 8$ for a destination d. If r receives an advertisement (d, 5) from its neighbor p, it will realize that there exists a better route that goes through p and has the hop count of $5 + 1 = 6$. The router r will update the table entry for d to reflect this improvement. Had r's initial hop count been less or equal 6, the advertisement would have been ignored, since it would not bring any improvement.

The value of hops must be an integer between 1 and 16, where 16 has the meaning of *infinity*. A destination with hops attribute set to 16 is considered unreachable. Hence, RIP will not be appropriate for systems that contain a router and a destination network that are more than 15 hops apart.

Appendix A.1 shows pseudo-code for RIP. A router advertises its routes by broadcasting RIP packets to all of its neighbors. A RIP packet contains a list of (destination, hops)-pairs. A receiving router compares its current metric for destination to (1+hops), which is the metric of the advertised route. The corresponding routing entry is updated if the new route is shorter. There is one exception to this rule—if the receiving router has the advertising router as nextR for the route, it adopts the alternative route regardless of its metric. We call this a *blind update*. Basically, this means that the previously advertised route has been overwritten by the advertising router.

Normally, a RIP packet contains information that reflects the advertising router's own routing table. This rule has an exception too. It is motivated by the observation that it is useless to advertise a route back to the interface through which it was learned. However, because of the periodic nature of the protocol, a router can not just stop advertising on a particular interface. Instead, if a route is learned over the interface i, it is advertised on that interface with hops set to 16 (infinity). This rule is called *split horizon with poisoned reverse* and its purpose is to prevent creation of small routing loops.

Each routing table entry has a timer expire associated with it. Every time an entry is created or

modified, the corresponding expire timer is re-set to 180 seconds. Routers try to advertise every 30 seconds, but due to network failures and congestion, some advertisements may not get through. If a route has not been "refreshed" for 180 seconds (i.e. no advertisement has been received for that destination from the next hop router), the router will assume that there has been a link failure, the destination will be marked as unreachable and a special garbageCollect timer will be set to 120 seconds. If this timer expires before the entry gets updated, the route is expunged from the table.

4.2 Standard vs. Theory

The mathematical theory behind RIP is described in [5] as the Distributed Bellman-Ford Algorithm. Two versions of the algorithm are presented—synchronous and asynchronous. In the synchronous version, updating happens in rounds. In each round, all the routers simultaneously receive advertisements from their neighbors, they select the best route and accordingly update their routing tables. It is not hard to prove that this algorithm always finds shortest routes. This algorithm does not have a great practical value, since it is synchronous. Today's networking technology does not permit global synchronization of events across a wide area network. This led to the design of the asynchronous version of the algorithm. In the Asynchronous Distributed Bellman-Ford Algorithm (ADBF for short), at every point in time, a router is either idle, sending an advertisement, or receiving an advertisement. The routing table is updated upon receipt of an advertisement. This introduces more nondeterminism in the ordering of updates, which consequently complicates the proof that shortest routes will be found. The proof uses, as an auxiliary result, the convergence theorem for the synchronous version. Details can be found in [5].

We first wanted to know whether we can easily adapt that proof to show that RIP protocol converges to the set of shortest routes. Our conclusion was negative. Although based on ADBF, RIP standard [23, 35] differs from it in several important details:

- ADBF has more powerful "bookkeeping" (i.e. it maintains a more elaborate state). In RIP, routers keep track of only one, currently best, route to each destination. On the other hand, ADBF nodes keep, for each destination, the most recent routes *through each of the neighbors*. Correspondingly, this would be reflected in the pseudo-code (Appendix A.1) by all subscript indices becoming *(dest,neighbor)*, instead of just *dest*. This makes ADBF more flexible, which comes at the expense of maintaining a larger data structure.
- RIP has blind updates. As a consequence of the previous difference, RIP routers need to separately handle the case when an advertisement is received from a neighbor which is already nextR for the route. This case is handled by blindly accepting the route. Contrary to that,

an ADBF router can in that case fall back to the second best route, which was previously advertised by some other neighbor. This difference shows a tradeoff between the speed of convergence and memory consumption.

- RIP's route length is bounded. RIP can handle routes of at most 15 hops. Distances of 16 or more hops are all considered equivalent to infinity. This optimization is introduced because it is known that in RIP-like protocols, maximum lifetime of loops is proportional to this upper bound on distances. In particular, there is no constant upper bound on the lifetime of loops in ADBF. Therefore, the optimization balances the tradeoff between quicker loop elimination and a wider range of route propagation.
- RIP has the *split horizon with poisoned reverse* rule mentioned in the previous section. This is another engineering optimization which is not present in ADBF.

The first of the above differences alone would suffice to make the proof of convergence for RIP substantially different from that of ADBF. Besides matching the RIP standard closely, our proof technique also gives useful insights about the speed of propagation of updates, which can be used for establishing timing bounds for convergence.

4.3 Convergence of RIP

In this section we present a proof of convergence for RIP. We prove that, in the absence of topology changes, RIP will find shortest routes to any destination, from every router inside the range of 15 hops from the destination.

We model the universe $\mathcal{U}$ as a bipartite connected graph whose nodes are partitioned into *networks* and *routers*, such that each router is connected to at least two networks. In other words, *routers* and *networks* are nodes, while *interfaces* are edges. Route computations involving different destination networks are independent, so we can assume that we are dealing with a single destination network d. A route entry for d at a router r consists of the three parameters:

- $\mathsf{hops}(r)$: current estimate of the distance metric to d (an integer between 1 and 16 inclusively).
- $\mathsf{nextN}(r)$: the next network on the route to d.
- $\mathsf{nextR}(r)$: the next router on the route to d.

Both r and $\mathsf{nextR}(r)$ must be connected to $\mathsf{nextN}(r)$. We say that r *points* to $\mathsf{nextR}(r)$. Initially, the universe must be in a *sound* state: that is, routers connected to d must have their metric set

to 1, while others must have it set to values strictly greater than 1. We can ensure this, since the routers will know which networks are initially connected to their interfaces. Two routers are *neighbors* if they are connected to the same network. The universe changes its state (i.e. routing tables) as a reaction to *update messages* being sent between neighboring routers. Each update message can be represented as a triple $(\mathsf{snd}, \mathsf{net}, \mathsf{rcv})$, meaning that the router snd sends its current distance estimate through the network net to the router rcv. This will cause the receiving router to update its routing entry. An infinite sequence of such messages $(\mathsf{snd}_i, \mathsf{net}_i, \mathsf{rcv}_i)_{i\geq 0}$ is said to be *fair* if every pair of neighboring routers s and r exchanges messages infinitely often:

$$\forall i.\ \exists j > i.\ (\mathsf{snd}_j = s) \ \text{ and } \ (\mathsf{rcv}_j = r).$$

This property simply assures that each router will continuously communicate its routing information to all of its neighbors.

On the outermost level, our proof uses induction on distance from the destination. *Distance* to d is defined as

$$D(r) = \begin{cases} 1, & \text{if } r \text{ is connected to } \mathsf{d} \\ 1 + \min\{D(s) \mid s \text{ neighbor of } r\}, & \text{otherwise.} \end{cases}$$

For $k \geq 1$, the k-*circle* around d is the set of routers

$$C_k = \{r \mid D(r) \leq k\}.$$

The key notion in our proof is that of the k-stability.

Definition: [Stability] For $1 \leq k \leq 15$, we say that the universe is k-*stable* if both of the following properties hold:

(S1) Every router $r \in C_k$ has its metric set to the actual distance: that is, $\mathsf{hops}(r) = D(r)$. Moreover, if r is not connected to d, it has its next network and next router set to the first network and router on some shortest path to d: that is, $D(\mathsf{nextR}(r)) = D(r) - 1$.

(S2) For every router $r \notin C_k$, $\mathsf{hops}(r) > k$. □

Given a k-stable universe, we say that a router r at distance $k+1$ from d is $(k+1)$-*stable* if it has an optimal route: that is, $\mathsf{hops}(r) = k+1$ and $\mathsf{nextR}(r) \in C_k$.

Our goal is to prove the following main theorem about correctness of RIP:

Theorem 1 (Correctness of RIP) *For any $k < 16$, starting from an arbitrary sound state of the universe $\mathcal{U}$, for any fair sequence of update messages, there is a time t_k such that $\mathcal{U}$ is k-stable at all times $t \geq t_k$.* □

Notice that besides convergence of RIP, this theorem also guarantees optimality of the routes. This is because the routes inside a stable circle are in fact optimal.

The proof will be carried out by induction on k. The following lemma provides the basis for induction.

Lemma 1 *The universe is initially 1-stable.* ☐

This is a consequence of the soundness of the initial state.

A key property of k-stability is that once it is achieved, it is preserved forever. This would not be true if our definition of stability did not contain condition (S2). This condition strengthens the induction hypothesis enough that we can induct on k-stability.

Lemma 2 (Preservation of stability) *For any $k < 16$, if the universe is k-stable at some time t, then it is k-stable at any time $t' \geq t$.* ☐

To prove that a k-stable universe eventually becomes $(k+1)$-stable, it suffices to show that every *router* at distance $(k+1)$ eventually becomes $(k+1)$-stable. This is the statement of the next lemma.

Lemma 3 *For any $k < 15$ and router r such that $D(r) = k+1$, if the universe is k-stable at some time t_k, and the sequence of advertisements is fair, then there is a time $t_{r,k} \geq t_k$ such that r is $(k+1)$-stable at all times $t \geq t_{r,k}$.* ☐

One of the key facts used in the proof of this lemma is fairness of the advertisement sequence. Without the fairness, neighbors of r would be allowed to simply stop advertising to r at any point. This would keep r's routing table unchanged and hence prevent it from ever becoming $(k+1)$-stable.

Finally, using the fact that there are only finitely many routers, we easily derive the Progress Lemma which proves our inductive step and hence the Theorem 1.

Lemma 4 (Progress) *For any $k < 15$, if the universe $\mathcal{U}$ is k-stable at some time t_k, and the advertisement trace is fair, then there is a time $t_{k+1} \geq t_k$ such that $\mathcal{U}$ is $(k+1)$-stable at all times $t \geq t_{k+1}$.* ☐

Our proof, which we call the *radius proof,* differs from the one described in [5] for the asynchronous Bellman-Ford algorithm. Rather than inducting on estimates for upper and lower bounds for distances, we induct on the the radius of the stable region around d. The proof has three attributes of interest:

1. It is automated.

2. It states a property about RIP, rather than ADBF.

3. The radius proof is more informative. It shows that correctness is achieved quickly close to the destination, and more slowly further away. It also implicitly estimates the number of advertisements needed to progress from k-stability to $(k+1)$-stability. We exploit this in the next section to show a real-time bound on convergence.

Notice, also, that our theorems do not say anything about formation of loops in RIP. Loops, indeed, can be created, but the correctness theorem guarantees that they will eventually be eliminated. In the worst case, loops can cause the behavior known as *counting to infinity*, which is described in the standard [23]. Even in that case, their lifetime is bounded by the convergence time, which is calculated in Section 4.4.

4.3.1 Proof Methodology

Lemma 1 is easily proved by HOL and serves as the basis of the overall induction. Lemma 2 is the fundamental safety property, which we proved both in HOL and in SPIN. To prove the lemma, one needs to show that a k-stable universe remains k-stable after an update message between any two neighbors. Our HOL proof proceeds by separately verifying that each of the conditions (S1) and (S2) remains true after the update. This can not be directly modeled in SPIN, since, for instance, the number of routers inside the k-circle is unknown. However, it turns out that k-stability gives rise to a nice *abstraction* of the system, which can be used to encode the system in SPIN. In this case, we know that in a k-stable universe, the k-circle always advertises the distance k to the outside world. Therefore, the k-circle can be modeled as a single router that always advertises the distance of k hops.

Observe that Lemma 3 only involves one router r at a distance $k+1$ from d. Starting from a k-stable state, we need to show that r converges to the correct value. Moreover, since all future states of the system are guaranteed to be k-stable (Lemma 2), r will receive advertisements from only two kinds of neighbors—those within the k-circle, and those outside it. However, all the distances that are advertised by the outside world are strictly greater than k. This leads to another abstraction, this time for the part outside the k-circle. Because of the k-stability, we can model it as a single router that nondeterministically advertises distances greater than k.

The two abstractions together provide a finitary representation of the system. We use it for proving Lemma 3 in SPIN. However, we first need to show that abstracting the system in this way indeed preserves the property that we are interested in—namely, Lemma 4. We proved this in HOL, by induction on the length of the fair advertisement trace. This proof is the crucial link

that allows us to join the HOL and SPIN results. The proof itself is rather complex with a large case analysis. However, the effort is justified since the finitary abstraction can now be used in multiple proofs with possible minor modifications. This re-use can be seen in the proof statistics in Table 5.1 at the end of Chapter 5. The abstraction proof is represented in the table as the HOL portion of the second proof of Stability Preservation (Lemma 2).

4.4 Sharp Timing Bounds for RIP Stability

In the previous section we proved convergence for RIP conditioned on the fact that the topology stays unchanged for some period of time. We now want calculate how long should that period of time be. To do this, we need to have some knowledge about the times at which certain protocol events must occur. In the case of RIP, we use the following assumption that describes the frequency of advertisements:

Fundamental Timing Assumption: There is a value Δ, such that during every topology-stable time interval of the length Δ, each router gets at least one update message from each of its neighbors. □

RIP routers normally try to advertise every 30 seconds; a failure to receive an advertisement within 180 seconds is treated as a link failure. Thus, $\Delta = 180$ seconds satisfies the Fundamental Timing Assumption for RIP. [1] We will assume (and not explicitly mention) this assumption in all of the subsequent theorems about RIP. Notice that the assumption implies fairness of the advertisement trace.

As in the previous section, we will concentrate on a particular destination network d. Our timing analysis is based on the notion of weak stability.

Definition: [Weak stability] For $2 \le k \le 15$, we say that the universe $\mathcal{U}$ is *weakly k-stable* if all of the following conditions hold:

(WS1) $\mathcal{U}$ is $(k-1)$-stable.

(WS2) $\forall r.\ D(r) = k \ \Rightarrow\ (r \text{ is } k{-}\text{stable or } \mathsf{hops}(r) > k)$.

(WS3) $\forall r.\ D(r) > k \ \Rightarrow\ \mathsf{hops}(r) > k$. □

Weak k-stability is stronger than $(k-1)$-stability, but weaker than k-stability. The second disjunct in (WS2) is what distinguishes it from the ordinary k-stability. This disjunction will introduce

[1] If the routers are required to perform triggered updates, Δ will be much smaller, typically around 5 seconds.

additional complexity in case analyses involving weak stability. As with the k-stability, we have the preservation lemma:

Lemma 5 (Preservation of weak stability) *For any $2 \leq k \leq 15$, if the universe is weakly k-stable at some time t, then it is weakly k-stable at any time $t' \geq t$.* □

The Fundamental Timing Assumption provides a connection between discrete advertisement events and continuous time. Precisely, to show that some property P holds after a Δ time interval, it is enough to prove that P holds after each router receives at least one advertisement from each of its neighbors. We use this technique in the subsequent lemmas.

First we show that the initial state inevitably becomes weakly 2-stable after one update interval.

Lemma 6 (Initial progress) *Starting from an initial state, the universe will become weakly 2-stable after Δ time.* □

The main progress property says that it takes one update interval to get from a weakly k-stable state to a weakly $(k+1)$-stable state. This property is shown in two steps. First we show that condition (WS1) for weak $(k+1)$-stability holds after Δ:

Lemma 7 *For any $2 \leq k \leq 15$, if the universe is weakly k-stable at some time t, then it is k-stable at time $t + \Delta$.* □

Then we show the same for conditions (WS2) and (WS3). The following puts all three steps together:

Lemma 8 (Progress) *For any $2 \leq k < 15$, if the universe is weakly k-stable at some time t, then it is weakly $(k+1)$-stable at time $t + \Delta$.* □

The *radius* of the universe (around d) is the maximum distance from d:

$$R = max\{D(r) \mid r \text{ is a router}\}.$$

The main theorem describes convergence time for a destination in terms of the radius around it:

Theorem 2 (RIP convergence time) *A universe of radius R becomes 15-stable within $max\{15, R\} \cdot \Delta$ time, assuming that there are no topology changes during that time interval.* □

The theorem is an easy corollary of the preceding lemmas. Consider a universe of radius $R \leq 15$. To show that it converges in $R \cdot \Delta$ time, observe what happens during each Δ-interval of time:

after Δ	weakly 2-stable	(by Lemma 6)
after $2 \cdot \Delta$	weakly 3-stable	(by Lemma 8)
after $3 \cdot \Delta$	weakly 4-stable	(by Lemma 8)
...	...	...
after $(R-1) \cdot \Delta$	weakly R-stable	(by Lemma 8)
after $R \cdot \Delta$	R-stable	(by Lemma 7)

R-stability means that all the routers that are not more than R hops away from d will have shortest routes to d. Since the radius of the universe is R, this includes *all* routers.

An interesting observation is that progress from k-stability to $(k+1)$-stability is not guaranteed to happen in less than $2 \cdot \Delta$ time (we leave this to the reader). Consequently, had we chosen to calculate convergence time using stability, rather than weak stability, we would get a worse upper bound of $2 \cdot (R-1) \cdot \Delta$. In fact, our upper bound is sharp: in a linear topology, update messages can be interleaved in such a way that convergence time becomes as bad as $R \cdot \Delta$. Figure 4.1 shows

Figure 4.1: Maximum convergence time

such an example that consists of k routers and has the radius k with respect to d. Router r_1 is connected to d and has the correct metric. Router r_2 also has the correct metric, but points in the wrong direction. Other routers have no route to d. In this state, r_2 will ignore a message from r_1, because that route is no better than what r_2 (thinks it) already has. However, after receiving a message from r_3, to which it points, r_2 will update its metric to 16 and lose the route. Suppose that, from this point on, messages are interleaved in such a way that during every update interval,

all routers first send their update messages and then all of them receive update messages from their neighbors. This will cause exactly one new router to discover the shortest route during every update interval. Router r_2 will have the route after the second interval, r_3 after the third interval, $\ldots$, and r_k after the k-th interval. This shows that our upper bound of $k \cdot \Delta$ is reachable.

4.4.1 Proof Methodology

Lemmas 5, 6, and 8 are proved in SPIN (Lemma 7 is a consequence of Lemma 8). The proofs are carried out using an abstraction similar to the one mentioned in the previous section, appropriately tuned for the weak stability instead of the ordinary stability. Theorem 2 is then derived as a corollary in HOL. SPIN turned out to be extremely helpful for proving properties such as Lemma 8, which involve tedious case analysis. To illustrate this, assuming weak k-stability at time t, let us look at what it takes to show that condition (WS2) for weak $(k+1)$-stability holds after Δ time. ((WS1) will hold because of Lemma 7, but further effort is required for (WS3).)

To prove (WS2), let r be a router with $D(r) = k+1$. Because of weak k-stability at the time t, there are two possibilities for r: (1) r has a k-stable neighbor, or (2) all of the neighbors of r have $\mathsf{hops} > k$. To show that r will eventually progress into either a $(k+1)$-stable state or a state with $\mathsf{hops} > k+1$, we need to further break the case (2) into three subcases with respect to the properties of the router that r points to: (2a) r points to $s \in C_k$ (the k-circle), which is the only neighbor of r from C_k, or (2b) r points to $s \in C_k$, but r has another neighbor $t \in C_k$ such that $t \neq s$, or (2c) r points to $s \notin C_k$. These cases are relevant, because they determine who can become the nextR of r, once r becomes weakly $(k+1)$-stable. Each of these cases, branches into several further subcases based on the relative ordering in which r, s and possibly t send and receive update messages.

Doing such proofs by hand can be difficult. Essentially, the proof is a deeply-nested case analysis in which *final* cases are straightforward to prove—an ideal task for a model checker! Our SPIN verification is divided into four parts accounting for differences in possible topologies. These differences arise from the case analyses similar to the one sketched above. Each part has a distinguished process representing r and another processes modeling the environment for r. An environment is an abstraction of the rest of the universe. It generates all message sequences that could possibly be observed by r. In order to simplify the model, our abstraction allows the environment to generate even some sequences that are not possible in reality. Such abstractions will be sound for *universal* properties, stating that something holds in *every* possible run of the system. SPIN considered more cases than a manual proof would have required, 21,804 of them altogether for Lemma 8, but it checked these in only 1.7 seconds of CPU time. Even counting

set-up time for this verification, this was a significant time-saver. The resulting proof is probably also more reliable than a manual one. The statistics are summarized in Table 5.1 at the end of Chapter 5.

4.5 Summary

RIP is one of the oldest distance vector routing protocols. It is based on the Asynchronous Distributed Bellman-Ford Algorithm. We prove that, in the absence of topology changes, RIP is guaranteed to converge towards the set of shortest paths. The proof is based on the notion of *k-stability* and is carried out using the theorem prover HOL and the model checker SPIN. We also establish a sharp real-time bound on convergence time which linearly grows with the network radius. This proof is based on the notion of *weak k-stability* and is also carried out by a combination of HOL and SPIN. The work is described in [6, 10].

Chapter 5

Ad Hoc On Demand Distance Vector Protocol (AODV)

5.1 Protocol Description

As pointed out in Chapter 2, loop freedom is an important requirement for ad hoc routing protocols. In addition to saving bandwidth, which is a critical resource in this case, loop freedom also avoids some of the unpleasant behaviors (e.g. counting to infinity), typical for distance vector protocols. There has been a variety of proposals for distance vector protocols that avoid loop formation. We consider AODV [45], a recently introduced protocol that aims to minimize the flow of control information by establishing routes only *on demand.* Our analysis is based on the version 2 draft specification [44].

Each AODV node maintains one route record for every currently active destination. This set of destinations changes depending on the network traffic. Typically, a node will maintain a route to some destination if it anticipates that it will initiate or forward traffic towards that destination in the near future. A route record to a destination d contains the following fields:

$\mathsf{next_d}$**:** Next node on a path to d.

$\mathsf{hops_d}$**:** Distance from d, measured in the number of nodes (hops) that need to be traversed to reach d.

$\mathsf{seqno_d}$**:** Last recorded *sequence number* for d.

$\mathsf{lifetime_d}$**:** Timer whose value denotes the remaining time before route expiration.

The purpose of sequence numbers is to track changes in topology. Each node maintains its own sequence number. It is incremented whenever the set of neighbors of the node changes. In addition to that, each route entry contains a sequence number of the destination *at the time the route is discovered*. As the topology changes, more recent routes will have higher sequence numbers. That way, nodes can distinguish between recent and obsolete routes.

When a node s wants to communicate with a destination d, it broadcasts a route request (RREQ) message to all of its neighbors. The message has the following format:

$$\mathsf{RREQ}(\mathsf{hops_to_src}, \mathsf{broadcast_id}, \mathsf{d}, \mathsf{seqno}, s, \mathsf{src_seq_no}).$$

- Argument $\mathsf{hops_to_src}$ determines the current distance from the node which initiated the route request. The initial RREQ has this field set to 0, and every subsequent node increments it by 1.

- Argument $\mathsf{broadcast_id}$ is the *broadcast ID* of the initiating router (s). RREQ messages are flooded through the network and *broadcast ID* is used to identify multiple copies of the same message. For instance, if a node receives two messages from the same source with the same broadcast ID, the second message is discarded. Every time before a node initiates a route request, it increments its broadcast ID.

- Argument seqno specifies the least sequence number for a route to d that s is willing to accept. Node s usually uses here the last sequence number it recorded for the destination, namely $\mathsf{seqno}_\mathsf{d}$.

- Argument $\mathsf{src_seq_no}$ is the sequence number of the initiating node.

When a node t receives an RREQ, it first checks whether it has a route to d marked with a sequence number at least as big as seqno. If it does not, it rebroadcasts the RREQ with incremented $\mathsf{hops_to_src}$ field. At the same time, t can use the received RREQ to set up a reverse route to s. This route would eventually be used to forward replies back to s. If t has a fresh enough route to d, it replies to s (unicast via the reverse route) with a route reply (RREP) message which has the following format:

$$\mathsf{RREP}(\mathsf{hops}_\mathsf{d}, \mathsf{d}, \mathsf{seqno}_\mathsf{d}, \mathsf{lifetime}_\mathsf{d}).$$

Arguments $\mathsf{hops}_\mathsf{d}, \mathsf{seqno}_\mathsf{d}$, and $\mathsf{lifetime}_\mathsf{d}$ are the corresponding attributes of t's route to d. However, if t is the destination itself ($t = \mathsf{d}$), it replies with

$$\mathsf{RREP}(0, d, \mathsf{big_seq_no}, \mathsf{MY_ROUTE_TIMEOUT}).$$

The value of big_seq_no needs to be at least as big as d's own sequence number and at least as big as seqno from the request. Parameter MY_ROUTE_TIMEOUT is the default lifetime, locally configured at d. Every node that receives an RREP increments the hops field of the packet and forwards it along the reverse route to s. When a node receives an RREP for some destination d, it uses the information from the packet to update its own route for d. If it already has a route to d, preference is given to the route with a higher sequence number. If sequence numbers are the same, the shorter route is chosen. This preference rule is used by everybody.

In addition to the routing table, each node s keeps track of the *active neighbors* for each destination d ($\mathsf{active_d}$). This is the set of neighboring nodes that use s as their $\mathsf{next_d}$ on the way to d. If s detects that its route to d is broken, it sends an unsolicited RREP message to all of its active neighbors for d. This message contains hops set to 255 (infinity), and seqno set to one more than the previous sequence number for that route. Because of the previously mentioned preference rule for route selection, this artificially incremented sequence number forces the recipients to accept this "route" (which is in fact a notification of the route breakage, since hops is set to infinity) and propagate it further upstream, all the way to the initial route requester(s).

If a route expires (i.e. $\mathsf{lifetime_d}$ generates a timeout event), it is marked as invalid, the corresponding hop-count is set to infinity and $\mathsf{lifetime_d}$ is re-set to BAD_LINK_LIFETIME, which is a locally configured constant. The route can be replaced or repaired during that time. If the timer expires while the route is still invalidated, the route entry is marked as "erasable". This means that it can be garbage collected as needed. We will soon see that this is a critical point for loop formation. When erasing an entry, the router "forgets" the sequence number for the corresponding destination. If this happens too soon, loops can be formed with nodes that have old, but still unexpired routes.

The pseudo-code for AODV is provided in the Appendix A.2. It is based on the version 2 of the standard [44]. However, the original standard specification was incomplete in three places: it did not specify anything about the initial state, it did not describe how to handle incoming RREP packets and it forgot to specify that routers should increment the sequence numbers *in their own routing tables* when sending unsolicited RREPs. It was fairly straightforward to fill in the missing gaps, so this is what we did first. This is reflected in the pseudo-code in Appendix A.2. This completed version of the protocol standard was our base for the rest of the analysis.

5.2 Loop Conditions

The question of central interest for us was whether sequence numbers protect AODV from forming loops. This was claimed to be the case and even a short formal argument in favor of this fact was presented in the original description of AODV [45]. However, the argument was somewhat superficial—it did not consider certain failure conditions that can leave the nodes in inconsistent states. As we will shortly see, loops can indeed be formed under these failure conditions.

Our first goal was to perform instance verification of the loop freedom conjecture using SPIN. Given the fact that sequence numbers are described in the standard as 32-bit integers, we anticipated a problem with a large state space. Because of that, we used a simple 3-node instance shown in Figure 5.1. Nodes A and B try to send data to the destination D. On top of each node is shown its current hop-count and sequence number for the route to D. Unfortunately, our prediction about a large state space was correct. Nevertheless, SPIN found a counter-example long before the state space was completely searched. As a matter of fact, SPIN found a number of different looping scenarios in different situations. They all arise because of the incorrect handling of the following events: route expiry, link breakage, packet loss and reboot.

A sample scenario is shown on Figure 5.2. The loop was formed because B deleted a route on expiry. Specifically, the problem was that B deleted the sequence number before A did. Generally, if deletion of sequence numbers is allowed, we have to make sure (if possible) that each node performs the deletion *before* its next hop. This was violated in the example just shown.

Another looping scenario is shown on Figure 5.3. This time B does not delete the route on expiry, but keeps it around as an expired route. It seems that the loop was formed because B did not increase its sequence number for D.

Figure 5.4 shows a scenario where, on route expiry, B first increases the sequence number for D and then sends an unsolicited RREP (error notification) to A. B assumes that it is now safe to garbage collect the route entry and it does so. This looks like a sound strategy in all cases except one—the case where A does not receive the unsolicited RREP. The figure shows how a loop can be created in that case.

A similar situation can happen as a result of a link breakage, rather than a route expiry. This is depicted on Figure 5.5.

Finally, Figure 5.6 shows the situation where a loop is formed because A did not notice that B was rebooted.

A more detailed discussion of the possible looping scenarios and their relationship to the standard can be found in [6].

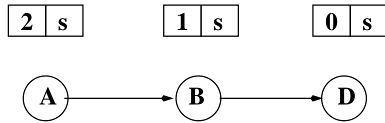

Figure 5.1: Initial configuration

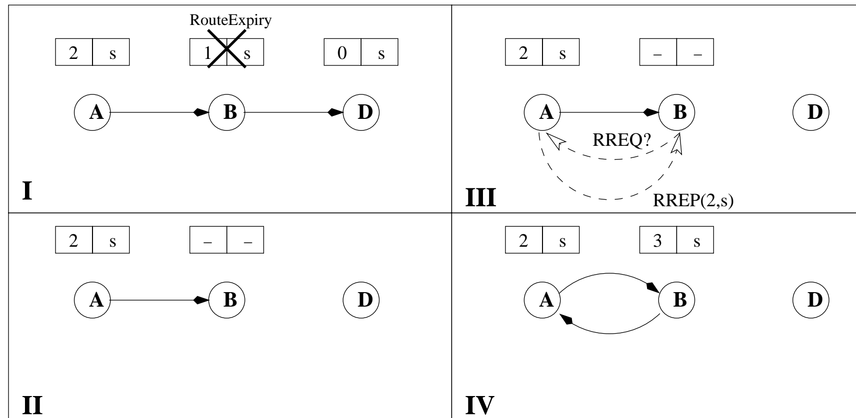

Figure 5.2: Looping scenario

5.3 Path Invariants and Loop Freedom

Guided by the discovered scenarios, we describe the assumptions under which we claim and prove that AODV will produce and maintain loop-free routes.

We can see that three out of the five described looping scenarios are ultimately caused by nodes losing the sequence number information too early. One is caused by a missing increment of the sequence number for a route and the remaining one by a "silent" reboot. We summarize these insights in the following three recommendations for modifying the AODV protocol:

A1. When a node discovers that its route to a destination has expired or broken, it increments the sequence number for the route.

A2. Nodes *never* delete routes.

A3. Nodes *always* detect when a neighbor restarts its AODV process. The restart is treated as if all links to the neighbor have been temporarily broken.

We need to modify the AODVv2 pseudo-code in accordance with these assumptions. A1 and A2 can be implemented by changing the handler for route expiry. This is shown in the Appendix A.3.

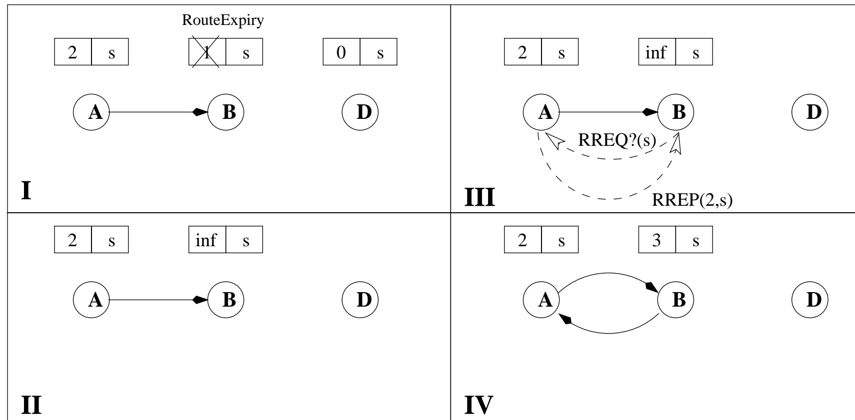

Figure 5.3: Looping scenario

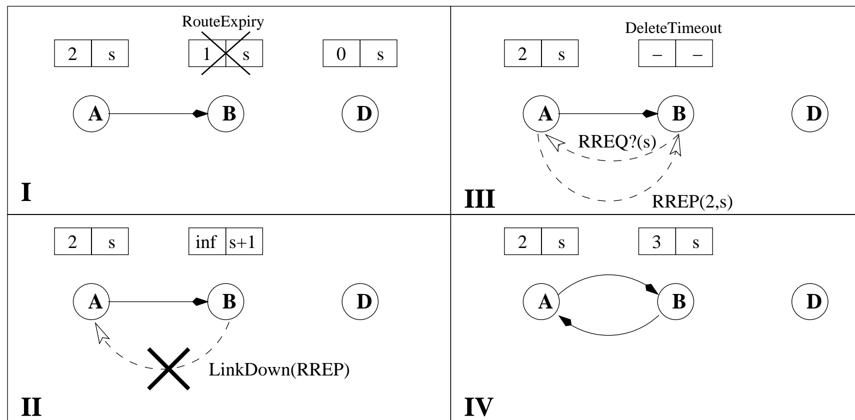

Figure 5.4: Looping scenario

The last assumption is environmental and is not reflected in the code. We expect A3 to be implemented through some link layer mechanism or by exchanging periodic "hello" messages. Hereafter, we will analyze this modified version of AODV.

The presented analysis of looping conditions for AODV is another example of the benefits of automated reasoning. It is unlikely that any manual method would provide these insights so systematically and so quickly. At best, the results would be derived through several unsuccessful attempts at proving loop freedom.

Now we can state our main theorem:

Theorem 3 (Loop-freedom of AODV) *Consider an arbitrary network of nodes running AODV. If all nodes conform to the assumptions A1-A3, there will be no routing loops formed.*

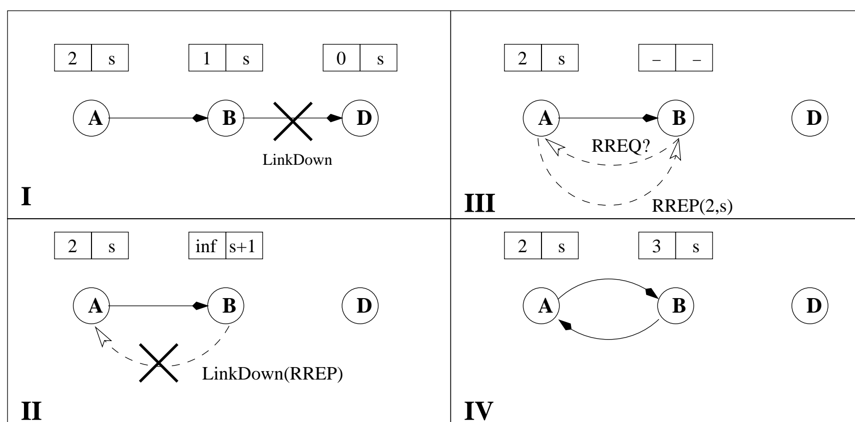

Figure 5.5: Looping scenario

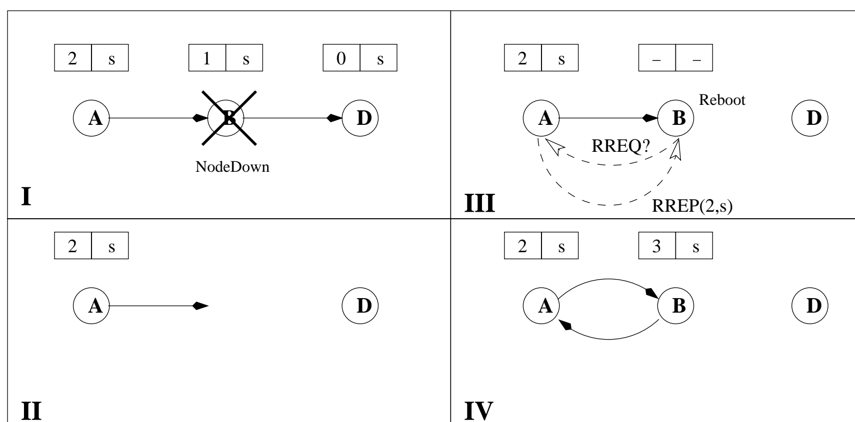

Figure 5.6: Looping scenario

We provide a completely automated proof of Theorem 3 using the SPIN model checker and HOL theorem prover. Our proof is based on a key path invariant of the protocol. This invariant is also used to prove route validity—the fact that following $\mathsf{next_d}$ pointers necessarily takes us to d, in one or more steps.

For arbitrary nodes n and d, we write $\mathsf{seqno_d}(n)(t)$ to denote n's sequence number for the destination d at the time t. Similarly for hops and next. In non-temporal properties we omit the time argument, understanding that we are talking about current values at some given time.

The following is an invariant (over time) of the AODV process at a node n, for every destination d:

Theorem 4 *If* $\mathsf{next_d}(n) = n'$, *then*

1. $\mathsf{seqno_d}(n) \leq \mathsf{seqno_d}(n')$, *and*

2. $\mathsf{seqno_d}(n) = \mathsf{seqno_d}(n') \Rightarrow \mathsf{hops_d}(n) > \mathsf{hops_d}(n')$. ☐

The theorem says that the pair $(-\mathsf{seqno_d}, \mathsf{hops_d})$ strictly decreases in the lexicographic ordering when a $\mathsf{next_d}$ pointer is followed. This invariant has two important consequences:

1. (Loop-Freedom) Consider the network at any instant and look at all the routing-table entries for a destination d. Any data packet traveling towards d would have to move along the path defined by the $\mathsf{next_d}$ pointers. We know from Theorem 4 that at each hop along this path, either the sequence number must increase or the hop-count must decrease. In particular, this implies that a node cannot occur twice on the route to d. This guarantees loop-freedom for AODV.

2. (Route Validity) Loop-freedom in a finite network guarantees that data paths are finite. This does not guarantee that the path ends at d. However, if all the sequence numbers along a path are the same, hop-counts must strictly decrease (by Theorem 4). In particular, the last node n_l on the path cannot have the hop-count equal to INFINITY (no route). But since n_l does not have a route to d, it must be equal to d.

To prove Theorem 4, we first prove the following lemmas about the routing table at each node n, now considered as a function of time.

Lemma 9 *If* $t_1 \leq t_2$, *then*

$\mathsf{seqno_d}(n)(t_1) \leq \mathsf{seqno_d}(n)(t_2)$. ☐

Lemma 10 *If* $t_1 \leq t_2$ *and*

$\mathsf{seqno_d}(n)(t_1) = \mathsf{seqno_d}(n)(t_2)$, *then*

$\mathsf{hops_d}(n)(t_1) \geq \mathsf{hops_d}(n)(t_2)$. ☐

Intuitively, Lemma 9 states that the sequence number for a given destination (d) at a given node (n) never decreases over time. Lemma 10 says that if the sequence number stays unchanged over some period of time, then the hop-count can only improve or remain the same during that period.

Suppose $\mathsf{next_d}(n)(t) = n'$. Then we define lut (*last update time*), to be the last time before t, when $\mathsf{next_d}(n)$ *changed* to n'. We claim that the following lemma holds for times t and lut:

Lemma 11 *If* $\mathsf{next_d}(n)(t) = n'$, *then*

1. $\mathsf{seqno_d}(n)(t) = \mathsf{seqno_d}(n')(\mathsf{lut})$, *and*

2. $\mathsf{hops_d}(n)(t) = 1 + \mathsf{hops_d}(n')(\mathsf{lut})$. □

The lemma essentially describes the way node n updated its routing table at time lut to point to n'.

It is easy to see that the three lemmas together imply Theorem 4. First, Lemmas 9 and 11 applied to lut and t yield

$$\mathsf{seqno_d}(n)(t) = \mathsf{seqno_d}(n')(\mathsf{lut}) \leq \mathsf{seqno_d}(n')(t),$$

which is the first part of Theorem 4. Furthermore, if the equality holds above, then we have

$$\mathsf{hops_d}(n)(t) - 1 = \mathsf{hops_d}(n')(\mathsf{lut}) \geq \mathsf{hops_d}(n')(t)$$

because of Lemmas 10 and 11. This shows that $\mathsf{hops_d}(n)(t) > \mathsf{hops_d}(n')(t)$, which is the second part of Theorem 4. This suffices to guarantee loop freedom (Theorem 3).

5.3.1 Proof Methodology

We model AODV in SPIN by a Promela process for each node. As described earlier, each process needs to maintain state in the form of a broadcast ID, a sequence number and a routing table. The process needs to react to several events, possibly updating this state. The main events are neighbor discovery, data or control (RREP/RREQ) packet arrival and timeout events like route expiration. It is relatively straightforward to generate the Promela processes from the pseudo-code.

In order to use SPIN to prove loop freedom under the conditions A1-A3, we first needed to reduce the problem to a finite-state verification. In the case of RIP, we did it by constructing a finitary property-preserving abstraction of the system and then proving the *original* property on the abstracted system. In the case of AODV, we take a different approach. We reduce the *property* (and not the model) to an invariant on pairs of nodes (Theorem 4). By doing so, we reduced an unbounded n-node verification to a 2-node verification. However, this new model still has a very large state space, because of the 32-bit sequence numbers. We solved this problem by introducing an abstraction for sequence numbers at every node. Instead of using the actual sequence numbers, we only need to use the fact that the advertised sequence number is smaller/equal/larger than the current one. This way, we effectively scale down the representation of sequence numbers from 32 to 2 bits.

Now we prove the three lemmas in SPIN. Each verification involves at most two AODV processes reacting to events produced by an environment of AODV routers. Lemmas 9 and 10 can be proved without restrictions on the events produced by the environment. Lemma 11 is trickier and requires the model to record the incoming $\mathsf{seqno_d}(n')$ and $\mathsf{hops_d}(n')$ whenever the protocol decides to change

Table 5.1: Protocol verification effort

Task	HOL	SPIN
Modeling RIP	495 lines, 19 defs, 20 lemmas	141 lines
Proving Lemma 2 Once	9 lemmas, 119 cases, 903 steps	
Proving Lemma 2 Again	29 lemmas, 102 cases, 565 steps	207 lines, 439 states
Proving Lemma 3	Reuse Lemma 2 Abstractions	285 lines, 7116 states
Proving Lemma 5	Reuse Lemma 2 Abstractions	216 lines, 1019 states
Proving Lemma 6	Reuse Lemma 2 Abstractions	221 lines, 1139 states
Proving Lemma 8	Reuse Lemma 2 Abstractions	342 lines, 21804 states
Modeling AODV	95 lines, 6 defs	302 lines
Proving Lemma 9		173 lines, 5106 states
Proving Lemma 10		173 lines, 5106 states
Proving Lemma 11		157 lines, 721668 states
Proving Theorem 4	4 lemmas, 2 cases, 5 steps	

$\mathsf{next}_{\mathsf{d}}(n)$ to n'. This is easily done by the addition of two variables. Subsequently, Lemma 11 is also verified by SPIN.

Finally, the proof that the three lemmas together imply Theorem 4 involves standard deductive reasoning, outlined in the previous subsection. That part was done using the HOL theorem prover. The overall verification statistics is given in Table 5.1.

5.4 Summary

AODV is an emerging standard for routing in mobile ad hoc networks. Because of the dynamic topology in such networks, routes have short lifetime, so AODV establishes them on demand. AODV uses *sequence numbers* to protect against formation of routing loops. We use SPIN to discover several scenarios that create routing loops in AODVv2. We then propose modifications to the protocol and prove that they guarantee loop freedom. The proof is based on a path invariant which demonstrates a function on local states whose value strictly decreases across any hop of any route produced by AODV. Since the invariant depends only on two nodes, it is checked using SPIN.

Chapter 6

Border Gateway Protocol (BGP)

6.1 Introduction

Routing protocols discussed so far are used for routing inside a single Autonomous System (AS). Contrary to that, the role of BGP is to establish paths *between* different AS's. Interior gateway protocols such as RIP and OSPF alone would not suit the purpose of global routing for at least two reasons: scalability and rigidness of routing policies. Indeed, the routing tables would grow to an unmanageable size and the smallest changes in topology would need to be propagated across great distances, even though they would probably not affect local routing decisions. At the same time, fixed "shortest-route-first" preference policies used in distance vector routing would not be satisfactory on a global scale. For instance, rules that always prefer 5-hop routes over 15-hop routes are more meaningful in a single autonomous system then in a global internetwork. This is because network elements inside a single AS are likely to be similar, so the *number* of hops alone can be used to quite accurately compare the latency of different routes. Contrary to that, a global network can have "short" routes (measured in the number of hops) that are slower than some "long" routes. Moreover, route preference may be affected by other (sometimes proprietary) factors, such as security considerations. These factors are not very important inside a single AS, since the whole AS is managed by a single network administrator. All this points to the need for a more flexible and independent routing architecture.

One solution that would allow maximum flexibility in configuring routing policies is *source routing*. It allows every router to choose an arbitrary path to every destination, independently of the paths chosen by its neighbors. However, the Internet widely uses the "hop-by-hop" routing

paradigm in which every router makes a decision only about the next hop towards a given destination. BGP is a "hop-by-hop" protocol in which routers base their next hop decisions on the *entire* path to the destination. In that sense, BGP is a hybrid between source routing and "hop-by-hop" protocols.

In the remaining part of this chapter, we first give a description of the protocol based on [28, 48], and its formalization used in [19, 21]. We use this formalization to demonstrate the possibility of a slow *convergent* behavior. We then develop a real-time model of BGP and prove a fundamental theorem about the convergence time. We finally illustrate the power of the theorem by deriving several interesting results as corollaries.

6.2 Protocol Description

BGP routers exchange routing information during BGP sessions. Each session involves exactly two endpoint routers which are called *BGP peers*. BGP does not have its own reliability mechanisms, but instead uses a reliable transport protocol TCP. This ensures orderly delivery of messages, detects duplicates, recognizes when information has been lost, etc. Four kinds of messages are used in BGP sessions:

OPEN message is sent immediately after the TCP connection has been established. Its purpose is identification and mutual agreement about values of some protocol parameters, like timers.

KEEPALIVE messages are exchanged periodically to indicate that the peer on the other side is still up and running.

UPDATE messages contain routing information.

NOTIFICATION messages are used to indicate errors that have occured during the BGP session.

When both routers commit to the session, they may start exchanging routing information. To improve the efficiency of routing, a single autonomous system may deploy several BGP routers. If that is the case, these routers can use BGP in both the inter-domain (external) and the intra-domain (internal) mode. External usage of BGP is called E-BGP and the internal usage is called I-BGP. Routes to other AS's are first learned from some external router, via E-BGP. After that, I-BGP can be used to disseminate them between other routers in the same AS.

An important property of BGP is that once advertised, routes do not need to be refreshed. They stay active until they are explicitly revoked or until the TCP connection breaks. When that happens, each router must stop using the information it learned from the other one.

Withdrawn Routes
Path Attributes
Network Layer Reachability Information (NLRI)

Figure 6.1: UPDATE message

An UPDATE message consists of three parts, as shown on Figure 6.1. The Withdrawn Routes field lists the destinations for which the sending router is no longer ready to forward packets. In BGP, destinations are encoded as prefixes of IP addresses. A prefix is usually written in the format

$$\langle \text{IP address} \rangle / \langle \text{number of significant bits} \rangle.$$

For instance, the prefix `132.72.0.0/21` matches any IP address whose first 21 bits are identical to the first 21 bits of the address `132.72.0.0`.

The NLRI field lists destinations (IP prefixes) which the sender of the UPDATE message can reach using some route. The actual route, together with additional attributes is given in the Path Attributes field. It is assumed that these attributes apply to *all* the destinations in the NLRI field.

The exact set of path attributes that an UPDATE message contains may vary. Generally, attributes can be *well-known* or *optional*. Well-known attributes must be recognized by all BGP implementations. Besides them, a path may contain a number of optional attributes. They are not specified by the BGP standard and each router uses them at its own convenience, to communicate additional information about the path. Well-known attributes can be further subdivided into *mandatory* and *discretionary* attributes. Mandatory attributes must be included in every UPDATE message, while discretionary attributes may or may not be included. Below are examples of some of the well-known attributes:

AS_PATH is a well-known mandatory attribute that describes a route. It is encoded as a list of path segments, where each segment is either

- an AS_SET, which contains an unordered collection of ASNs [1], or
- an AS_SEQUENCE, which contains an ordered collection of ASNs.

In practice, most routes are advertised as a single AS_SEQUENCE. The purpose of AS_SET is to allow route aggregation. By using aggregation, a router can combine different routes into a single advertisement and thus reduce the amount of bandwidth used for exchanging control information.

[1] ASN=Autonomous System Number. It uniquely identifies an autonomous system.

LOCAL_PREF is a well-known discretionary attribute that describes the sender's degree of preference for the advertised route. Each router has locally configured policies that determine this level of preference. The policies play a crucial role in the route selection process. The LOCAL_PREF attribute is sent only to other BGP peers in the *same* AS.

NEXT_HOP is a well-known discretionary attribute which contains the IP address of the router that should be used as the next hop to the destinations listed in the UPDATE message. This is usually the advertising router itself, but in some cases it can be some other (possibly a non-BGP) router from the advertising router's AS.

MULTI_EXIT_DISCRIMINATOR (MED) is a well-known discretionary attribute. Its purpose is to discriminate between multiple exit or entry points to the same neighboring AS. The value of this attribute describes a certain metric, so that routes with smaller MEDs will be preferred. Below we give one example of the usage of MED.

Consider the situation shown in Figure 6.2. There are four AS's and four BGP sessions (depicted as links). AS2 would like to inform AS1 that it should favor Link A for traffic towards AS3 and Link B for traffic towards AS4. Notice that AS2 advertises to AS1 routes for AS3 and AS4 on both links (A and B). In order to signal this preference to AS1, AS2 should advertise a route for AS3 with higher MED value on Link B and lower MED value on Link A. The situation is the opposite for routes to AS4: higher MED on Link A and lower MED on Link B.

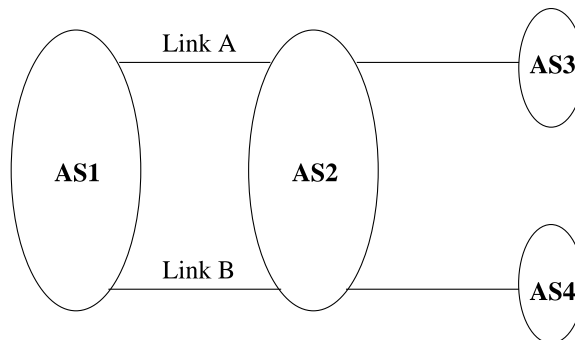

Figure 6.2: An example of the usage of MED

After an UPDATE message is received with all its attributes, the receiving router needs to process it. Conceptually, every BGP router has its routing table split into three parts, usually called RIBs (Routing Information Bases):

Adj-RIB-In . There is one Adj-RIB-In per peer—it stores routes that have been learned from that peer.

Loc-RIB contains routes that were selected for the router's own use.

Adj-RIB-Out . There is one Adj-RIB-Out per peer—it stores routes that are about to be advertised to that peer.

Adj-RIB-Ins contain unprocessed routing information that has been advertised to the router. A special decision process selects from Adj-RIB-Ins the routes which will be put in the router's own Loc-RIB and eventually disseminated to other peers. The process starts by computing the degree of preference for each route in Adj-RIB-In. The degree of preference is computed using the local router's preconfigured *policy*. The exact nature of this policy is not specified. Basically, a policy can be any non-negative integer valued function which depends only on the attributes present in the Adj-RIB-In.

The router now selects the route with the highest degree of preference and stores it in Loc-RIB. It is possible that several routes to the same destination have the same degree of preference, so the router may need to look at other attributes to break the ties. Additional attributes are considered in the following order:

1. Routes with shorter AS_PATH attributes are preferred.

2. Routes with smaller values of the MED attribute are preferred. [2]

3. Routes with shorter internal distance to the NEXT_HOP are preferred. This distance can, for instance, be learned from the interior routing protocol.

4. BGP can run in the external (E-BGP) and the internal (I-BGP) mode. In this step, the preference is given to the routes learned through E-BGP.

5. The final tie is broken by selecting the route advertised by the peer with the lowest BGP identifier number. [3]

Whenever the router's Loc-RIB changes, the changes need to be propagated to all its peers. However, if the received UPDATE message caused no changes in the Loc-RIB, no further propagation is needed. Also, whenever a new BGP router joins by establishing a BGP session, the other peer needs to advertise to the new router all the information from its Local-RIB.

[2] A router can be configured to ignore MEDs and therefore skip this step.

[3] Since several routers from the same AS can be running BGP at the same time, BGP identifiers are needed to distinguish between them.

BGP version 4 uses five timing parameters. Below are their names, interpretations and values suggested by the standard.

ConnectRetry is used during the initial phase of a BGP session, when the router is trying to establish a peering connection. It represents the maximum amount of time that a transport connection request to the other router can "hang" without being answered. Suggested value is 120 seconds.

HoldTime denotes the maximum amount of time that can elapse between two consecutive messages (of the type KEEPALIVE or UPDATE) from a given peer. If the peer has been silent for longer than HoldTime, an error NOTIFICATION message is sent and the connection is reset. Suggested value for HoldTime is 90 seconds.

KeepAlive determines the frequency of KEEPALIVE messages sent to the peer. The value of KeepAlive is the time between two successive messages. KEEPALIVE messages must not be sent more frequently than one per second. The suggested value is 30 seconds.

MinRouteAdvertisementInterval denotes the minimum amount of time that must elapse between route advertisements for a particular destination from the peer on the other end. The suggested value is 30 seconds.

MinASOriginationInterval denotes the minimum amount of time that must elapse between successive advertisements that report changes within the advertising router's own AS. The suggested value is 15 seconds.

6.3 Convergence of BGP

The BGP standard allows routers to have very flexible policies for route preference. A router sees the entire route when computing the degree of preference. Yet, the router's decision will determine only the next hop and not the entire path to the destination. In other words, a router's choice of a route must be a one-hop extension of one of its neighbors' choices. On the other hand, there are no similar dependencies on the policies level. As we will shortly see, this asymmetry can cause problems with convergence of BGP.

We first need to describe what exactly do we mean by *convergence of BGP*. Consider a collection of AS's, each with one or more BGP routers. Each router's policy is individually configured. We start BGP sessions between some of the routers and assume that the network topology does not changes during the run of the protocol. There are two possible behaviors of the system:

- Eventually, all the routers reach the point where they can not further improve their routes by extending the routes of their neighbors. When this happens, we say that the system has *converged* to a stable state.
- Routers keep changing their routing tables (and thus advertising new routes) forever. In this case we say that the system *diverges*.

Ensuring convergence is an important goal of most routing systems. When the system diverges, routers perpetually change their choice of routes. This extra processing can cut into router's performance, but more importantly, it can increase the frequency of inconsistent states which lead to packet loss. On the other hand, stable routes provide many other advantages, like better proxy caching and more stable loads.

It turns out that BGP can exhibit both convergent and divergent behaviors. Convergence crucially depends on the choice of routing policies. Generally, we say that a routing protocol is *safe* if it always converges in the absence of topology changes. RIP is an example of a safe protocol. Currently, BGP is not a safe protocol, even though it appears to behave safely in most practical instances. Several studies, both empirical and formal, have demonstrated anomalies associated with BGP behavior.

Labovitz et al. [32, 33] analyzed statistical occurrences of several unexpected scenarios and tried to explain their origins. They found a surprisingly large percentage of routing information that was useless and even "pathological".

Paxson [43] uses experimental data to study various properties of the BGP-produced routes, such as symmetry, persistence and prevalence. He found that Internet paths are heavily dominated by a single prevalent route, but lifetimes of such routes can vary from seconds to days. It is not clear to what extent this phenomenon is caused by the inefficiency of BGP as opposed to the pure dynamics of the Internet topology.

Villamizar et al. [53] suggest a method for "route flap damping". The idea in their approach is to slow down the propagation of potentially unstable routes. While this does not eliminate route oscillations, it reduces their practical impact on performance.

Varadhan et al. [52] use a formal model to derive a condition which characterizes convergent configurations for a special class of topologies—rings.

Griffin and Wilfong [20] described a formal model of BGP and proved that deciding convergence in that model is NP-hard. In [19], Griffin et al. demonstrated a condition sufficient for guaranteeing convergence. The condition is based on the notion of "dispute wheels", whose existence indicates potential (but not necessary) divergence. Finally, in [21], Griffin and Wilfong show a possible

extension of the model that is safe. The extension explores a tradeoff between stability and route preference. Convergence is achieved by pruning off certain potentially problematic routes. It is important to note that this solution is dynamic, unlike their previous static analysis. Static analysis can possibly reveal convergence problems, but it can not cure them. Moreover, there are potential problems with scalability, since deciding convergence is NP-hard. Finally, it is often not even possible to carry out a static analysis, because BGP policies may be proprietary.

For the rest of this chapter, we will focus on formal approaches to studying BGP convergence and its speed.

6.4 The Stable Paths Problem and the Simple Path Vector Protocol

This section describes a formalization of BGP which was used in [19, 21]. It consists of two parts:

The Stable Paths Problem (SPP) provides a formal semantics for BGP policies.

Simple Path Vector Protocol (SPVP) describes a simple formal version of the BGP protocol.

Intuitively, SPVP is a distributed algorithm that attempts to solve the Stable Paths Problem just like, say, Dijkstra's algorithm solves the Shortest Paths Problem. The analogy is not complete, as SPVP may actually fail to solve the Stable Paths Problem, even when there is a solution. [4]

A network is represented as a simple undirected connected graph $G = (V, E)$, where $V = \{0, 1, \ldots, n\}$ is the set of nodes connected by edges from E. We assume that nodes represent routers and edges represent BGP sessions. For a node u, its set of *peers* is $\text{peers}(u) = \{v \mid \{u, v\} \in E\}$. In most routing protocols, computations involving routes to different destinations are non-interfering. This allows us to assume, in our reasoning, that we are dealing with a single destination. We will assume that the node 0 is the destination to which all other nodes are trying to establish paths. A *path* in G is a sequence of nodes $(v_k\ v_{k-1}\ \ldots v_0)$, such that $\{v_i, v_{i-1}\} \in E$, for all $i, 1 \leq i \leq k$. For some technical reasons, we will assume the existence of a special *empty path*, denoted by ϵ. The empty path will be used to indicate the fact that a router does not have a proper path to the destination. Nonempty paths $P = (v_1\ v_2\ \ldots\ v_k)$ and $Q = (w_1\ w_2\ \ldots w_m)$ can be concatenated in a natural way if $v_k = w_1$. In that case, we define

$$PQ = (v_1\ v_2\ \ldots\ v_k\ w_2\ \ldots\ w_m).$$

[4] We will shortly see that a Stable Paths Problem does not necessarily have a solution.

For every path P, concatenation with the empty path produces the empty path:

$$P\epsilon = \epsilon P = \epsilon.$$

A path is called *simple* if it does not contain multiple instances of the same vertex. We will be almost exclusively interested in simple paths. For a simple path $P = (v_1\ v_2\ \ldots\ v_k)$ and any two of its nodes $u = v_l$ and $w = v_m$ $(l \leq m)$, we denote by $P[u \ldots w]$ the corresponding sub-path $(v_l\ v_{l+1}\ \ldots\ v_m)$.

For each $v \in V - \{0\}$, we denote by $\mathcal{P}^v$ the set of *permitted paths* from v to the destination (node 0). This is a subset of the set of all paths from v to 0, since a node may consider certain paths as absolutely unacceptable.

Let $\mathcal{P} = \{\mathcal{P}^v \mid v \in V - \{0\}\}$ be the set of all permitted path sets. For each $v \in V - \{0\}$, there is a *ranking function* $\lambda^v : \mathcal{P}^v \rightarrow \mathbf{N}$. For $P \in \mathcal{P}^v$, $\lambda^v(P)$ denotes the degree of preference that the node v gives to the path P. More preferable paths will have higher values of λ^v. Let $\Lambda = \{\lambda^v \mid v \in V - \{0\}\}$ be the set of all ranking functions.

Definition: [Stable Paths Problem] We say that a triple $S = (G, \mathcal{P}, \Lambda)$ is an instance of the *Stable Paths Problem* (SPP) if the following conditions are satisfied:

(SP1) Empty path is permitted: $\epsilon \in \mathcal{P}^v$.

(SP2) Empty path is lowest ranked: $\lambda^v(\epsilon) = 0$.

(SP3) Strictness: If $\lambda^v(P_1) = \lambda^v(P_2)$, then either $P_1 = P_2$, or $P_1 = (v\ u)P_1'$ and $P_2 = (v\ u)P_2'$ for some node u. In other words, P_1 and P_2 have the same next hop.

(SP4) Simplicity: If $P \in \mathcal{P}^v$, then P is a simple path (i.e. P does not contain repeated nodes). □

Notice that the set of *all* permitted paths $\Psi = \bigcup_{v \in V} \mathcal{P}^v$ can be naturally regarded as a partially ordered set with the ordering relation defined as

$$P < Q \text{ if and only if } \exists v.\ \lambda^v(P) < \lambda^v(Q).$$

We can analogously define the $\leq$ relationship on paths:

$$P \leq Q \text{ if and only if } \exists v.\ \lambda^v(P) \leq \lambda^v(Q).$$

Let $S = (G, \mathcal{P}, \Lambda)$ be an instance of the SPP. Given a node u and a set of paths $W \subseteq \mathcal{P}^u$ with distinct next hops, we define the *maximal path* in W with respect to u to be

$$\max(u, W) = \begin{cases} P \in W \text{ with maximal } \lambda^u(P), & \text{if } W \neq \emptyset \\ \epsilon, & \text{otherwise.} \end{cases}$$

A *path assignment* is a function π that maps each node $u \in V$ to a permitted path $\pi(u) \in \mathcal{P}^u$. In the BGP terminology, path assignments correspond to Loc-RIB routing tables. Given a path assignment π and a node u, we define the set of permitted one-hop extensions of neighboring paths as

$$\text{choices}(u, \pi) = \{(u\ v)\pi(v) \mid \{u, v\} \in E\} \ \cap \ \mathcal{P}^u.$$

We say that the path assignment π is *stable at node u* if

$$\pi(u) = \max(u, \text{choices}(u, \pi)).$$

The path assignment π is *stable* if it is stable at every node $u \in V$.

An SPP instance $S = (G, \mathcal{P}, \Lambda)$ is *solvable* if there exists a stable path assignment π for S. We call every such assignment a *solution* for S and write it as $(P_1, P_2, \ldots, P_n)$, where $\pi(u) = P_u$. An instance of SPP may have zero, one or more solutions. Consider the examples on Figure 6.3. [5] Permitted paths are listed next to each node in the order of preference - from the most preferred path on the top to the least preferred one on the bottom. It is easy to see that GOOD GADGET has the unique solution

$$\pi = ((1\ 3\ 0), (2\ 0), (3\ 0), (4\ 3\ 0)).$$

On the other hand, BAD GADGET is unsolvable. FLAP GADGET has two solutions:

$$\pi_1 = ((1\ 0), (2\ 1\ 0), (3\ 1\ 0), (4\ 3\ 1\ 0)), \text{ and}$$

$$\pi_2 = ((1\ 3\ 0), (2\ 0), (3\ 0), (4\ 3\ 0)).$$

Finally, NAUGHTY GADGET has the same unique solution π as GOOD GADGET, but SPVP may converge quite differently on those two SPP instances, as we will shortly see.

Notice that our model has a natural game-theoretic interpretation. [6] Each node represents a party in the game. Game positions correspond to path assignments. Available strategies of a player u are its peers. In a position π, a move of the node u consists of changing $\pi(u)$ to any path from the set choices(u, π). In particular, if v is a peer of u, then playing the "strategy" v means changing $\pi(u)$ to $(u\ v)\pi(v)$. Function λ^u evaluates the "yield" of the node u in a position π. It is easy to see that a path assignment π is stable if and only if π is a *Nash equilibrium* of this game. A Nash equilibrium is a position where no single node can increase its yield by switching to a different strategy.

Now we describe the Simple Path Vector Protocol (SPVP) from [21], whose pseudo-code is given in the Appendix A.4 [7]. Each node maintains two data structures:

[5] The examples are taken from [19, 21].

[6] A good overview of the basic game theoretic concepts can be found in [15].

[7] The pseudo-code given in the appendix describes SPVP in full, without the technical assumption that we deal with a single destination.

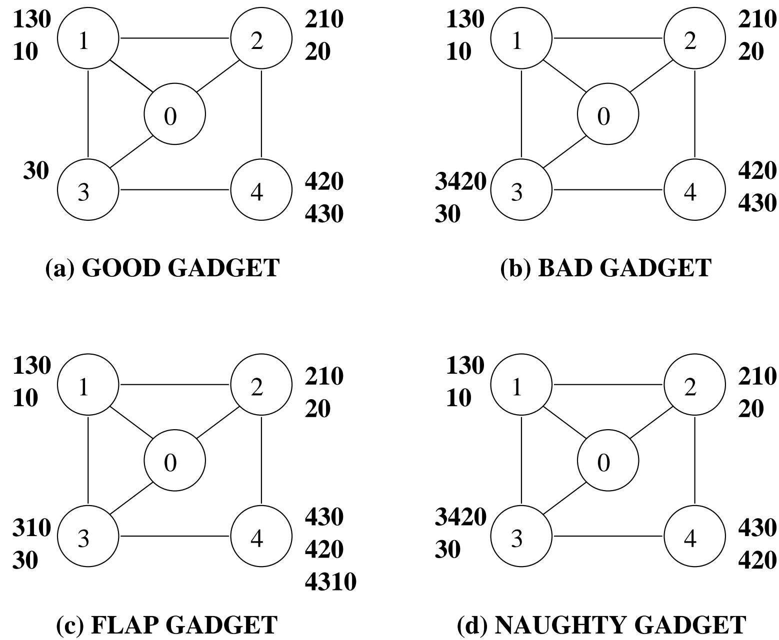

Figure 6.3: Examples of Stable Path Problems

- rib(u) is u's current path to the destination.
- rib_in($u \Leftarrow w$) denotes the path most recently processed at u, which was advertised by u's peer w.

Similarly as before, we define the set of path choices as

$$\text{choices}(u) = \{(u\ w)\text{rib_in}(u \Leftarrow w) \ \mid \ w \in \text{peers}(u)\} \ \cap \ \mathcal{P}^u,$$

and the best choice as

$$\text{best}(u) = \max(u, \text{choices}(u)).$$

Neighboring nodes keep exchanging paths that they have currently stored in the rib field. As the node u receives path advertisements from its peers, it uses them to recompute the set of available paths choices(u). The node tries to maintain the most preferred path from that set as its current path that will be stored in rib(u). Just as it is the case with BGP, when u changes its current path, it notifies its peers of that change. This may cause the peers to send advertisements to their peers, and so on. The process continues as long as there are unprocessed advertisements. .

We will now precisely define the protocol dynamics. Every node runs a copy of the SPVP process. Unprocessed advertisements are stored in reliable FIFO queues at the receiving end. Each node has one such queue for each peer—the queue where u stores advertisements from its peer w

is denoted by $\text{mq}(u \Leftarrow w)$. A *state* of the protocol is defined by states of all the routers (denoted by $\mathcal{S}$) and contents of all the queues mq (denoted by $\mathcal{Q}$). A router's state consists of the values of rib and rib_in fields. Therefore, a pair $(\mathcal{S}, \mathcal{Q})$ can be regarded as a global protocol state. Let $\pi = (P_1, P_2, \ldots, P_n)$ be any path assignment. An *initial* state induced by π is the state where for each router u, $\text{rib}(u) = P_u$, $\text{rib_in}(u \Leftarrow w) = \epsilon$, and each queue $\text{mq}(u \Leftarrow w)$ contains a single message P_w. [8] In other words, all the nodes have just advertised their current choices to their peers, but none of the advertisements have been processed yet. We assume that the protocol always starts in an initial state induced by some path assignment. The default initial assignment has all empty paths:

$$\forall u.\ P_u = \epsilon.$$

Activating a node r in a state $(\mathcal{S}, \mathcal{Q})$ proceeds in two steps:

1. The node first picks an unprocessed message. It does that by dequeuing the first message from one of the nonempty queues $\text{mq}(r \Leftarrow p)$.

2. The node then executes the body of the event handler *receive UPDATE(...) from p* given in the pseudo-code.

Executing the event handler can change r's local state—if the received route from p is preferred over the current route. If this happens, r will propagate that route by sending it to the queues of all of its neighbors. Therefore, the activation changes the global state to some new state $(\mathcal{S}', \mathcal{Q}')$. If all of r's queues were empty before the activation, the state remains unchanged. Generally, this defines a nondeterministic transition relation, written as

$$(\mathcal{S}, \mathcal{Q}) \xrightarrow{r} (\mathcal{S}', \mathcal{Q}').$$

An *activation sequence* is an infinite sequence of routers $(r_n)_{n \geq 1}$. We say that this activation sequence generates the state sequence $(\mathcal{S}_n, \mathcal{Q}_n)_{n \geq 0}$ if

$$(\mathcal{S}_{i-1}, \mathcal{Q}_{i-1}) \xrightarrow{r_i} (\mathcal{S}_i, \mathcal{Q}_i), \quad \text{for } i \geq 1.$$

We say that an activation sequence is *fair* if every router appears in it infinitely many times:

$$\forall r \in V - \{0\}.\ \forall i.\ \exists j > i.\ r_j = r.$$

The protocol stabilizes when all the queues become empty. After that point, the values of rib and rib_in fields would not change, so we say that the network is in a *stable* state. It is easy to show that in any stable state, the mapping $u \mapsto \text{rib}(u)$ defines a stable path assignment.

[8] If $w = 0$, we assume that $P_w = (0)$ (i.e. the destination advertises the trivial path for itself).

A natural question to ask is the following: "What is the connection between stable states of SPVP and solutions to the corresponding SPP?". We know that each stable state reached by SPVP induces a solution for the corresponding SPP. Consequently, we know that if an SPP instance S is unsolvable (such as BAD GADGET), SPVP necessarily diverges when we run it on S. However, the converse of this statement is not true. GOOD and NAUGHTY gadgets are good examples. Even though they have the same unique solution, it can be shown that SPVP *can* diverge when we run it on NAUGHTY GADGET, but it *always* converges on GOOD GADGET, as shown in [21]. This essentially shows that SPVP is a sound, but not a complete algorithm for solving SPP. It *can* diverge even on a solvable system. Whether this would happen, depends on the actual activation sequence.

Generally, policies can be set in an inconsistent way, so that not a single stable paths assignment exists. One way to isolate such possibly inconsistent policies is to look at *disputes* between them. Consider the peers u and v, and suppose that u wants to send its traffic to the destination via v. Suppose, furthermore, that v can choose between two paths to the destination: P and Q. In that case, u can also choose between $(u\ v)P$ and $(u\ v)Q$. The situation is depicted in Figure 6.4.

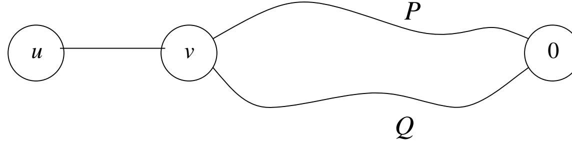

Figure 6.4: A potential dispute between policies

Problems can arise when u and v have the opposite preference for these paths. Precisely, we say that the pair of paths $(Q,\ (u\ v)P)$ forms a *dispute* if the following conditions hold:

1. Both P and Q are permitted at node v.

2. Path $(u\ v)P$ is permitted at node u.

3. $\lambda^v(P) \leq \lambda^v(Q)$.

4. Path $(u\ v)Q$ is not permitted at node u, or $\lambda^u((u\ v)Q) < \lambda^u((u\ v)P)$.

Given an SPP instance S, we can construct the corresponding *dispute digraph* $\mathcal{DD}(S)$. It is a directed graph whose nodes are permitted paths of S. There are two kinds of arcs:

- *Dispute arcs* are exactly those pairs of paths that form disputes.

- *Transmission* arcs are pairs of permitted paths of the form $(P,\ (u\ v)P)$, where u and v are peers.

A crucial result proved in [19] shows that convergence of SPVP is guaranteed when the corresponding dispute digraph is acyclic. More precisely, the theorem states that acyclicity of $\mathcal{DD}(S)$ implies that SPVP converges to a stable state under *any* fair activation sequence. We will provide an alternative proof of this fact in our real-time analysis of BGP convergence.

6.5 Measuring BGP Oscillation

So far, we have been concerned with the problem of convergent vs. divergent behavior of BGP. Convergence, however, is not the only important property. What we really want from a routing protocol is *fast* convergence. Knowing that routes will stabilize at some (unknown) point in the future does not mean much to the user. The user's ultimate goal is to have fast and reliable routing. Long periods of route instability, even if convergence is guaranteed to happen eventually, can adversely affect the performance. Moreover, the user may not even be able to tell the difference between a divergent and a slowly convergent behavior if the network topology changes frequently—convergence process is then restarted before it finishes. We clearly need to be able to measure the *speed* of convergence. This would allow us to perform a finer analysis and classify convergent systems into faster and slower.

Our first attempt is to approximate the convergence time by the maximum number of route oscillations that can happen on a single node.

Definition: [Oscillation index] Given an instance S of SPP, we define the *oscillation index of a router* r of S to be the maximum number of times that r can change its route to the destination during the convergence of SPVP. We denote that number by $OI(r, S)$. If there exists an activation sequence that causes r to never stabilize, we put $OI(r, S) = \infty$.
We define the *oscillation index of an instance* S as the maximum oscillation index of its nodes:

$$OI(S) = \max_{r \in S} OI(r, S).$$

$\square$

Intuitively, the oscillation index measures potential route instability during the convergence process. We immediately see that an SPP instance S can exhibit divergent behavior if and only if $OI(S) = \infty$. However, if the oscillation index is finite, it enables us to divide convergent instances into different classes based on their stability. Typically, we will measure the oscillation index for a family of instances as a function of the number of routers. If this function has a polynomial upper bound, we will regard the convergence as fast. For instance, we may have a specific set of instances

$\mathcal{S}$, such that for every $S \in \mathcal{S}$, we have $OI(S) \leq 3n^2$, where n is the number of routers in S. In that case we would say that SPVP convergence is fast on $\mathcal{S}$.

BGP is known to suffer from route instability. We would like to know the reason—is this due to the possibility of divergence, or is it the case that there are slow *convergent* behaviors as well? In other words, does convergence in BGP necessarily mean *fast* convergence? The following theorem states that in the general case, any upper bound on the oscillation index is at least factorial in the number of nodes!

Theorem 5 (Slow convergence of SPVP) *For every $n \geq 2$, there exists an n-node convergent SPP instance S_n such that $OI(S_n) \geq (n-2)!$.*

Proof: Let the set of routers be $V = \{0, 1, \ldots, n-1\}$, where 0 is the destination. We will prove a stronger statement: there exists a convergent SPP instance S_n with the n-clique topology (n nodes, every two are directly connected), where SPVP causes all non-destination nodes to go through all different simple paths to the destination. This suffices to prove the theorem, since the number of simple paths between any two nodes in an n-clique is at least $(n-2)!$.

We prove this by induction on n.

For $n = 2$ ($V = \{0, 1\}$), assume that S_2 is a 2-clique. Node 1 prefers the route (1 0) over ϵ, so it will eventually switch to the route (1 0), thus exploring the only path to the destination. This is a stable state and the base case is therefore proved, since node 1 has explored the only simple path to 0.

Assume that the statement holds for some n-clique S_n. Consider the $(n+1)$-clique S_{n+1} formed by adding the node n and edges to all the remaining nodes. The convergence has three phases:

- Phase I: Only nodes in S_n are activated in the order that causes them to go through all different paths that do not include n. This is possible according to the induction hypothesis. Node n is not activated in this phase, but it queues up all the advertisements from other nodes.

- Phase II: Node n is repeatedly activated till it consumes all the queued advertisements. We assume that every time n consumes a route, the route is preferred over the current route at n. This will cause the node n to go through all different paths to the destination and advertise them to all other nodes.

- Phase III: We again activate only nodes in S_n. This time, they will process all the queued advertisements from the node n. The advertisements would contain all the simple paths to the destination that do include the node n. Similarly as in Phase II, we assume that all the

nodes receive routes in the increasing order of preference. This will cause the nodes from S_n to explore all different paths that include the node n.

We conclude that during phases I and III, all non-destination nodes from S_n will go through all different paths to the destination. During Phase II, the node n will go through all different paths. Notice that nodes never go through the same route twice and that each node's route preference constantly increases. Because of this, the final state will be stable. □

The theorem tells us that even convergent BGP systems can converge very slowly. It is not quite clear whether $(n-2)!$ is actually an upper bound on the oscillation index of convergent n-node instances, but the figure is already alarmingly high. A similar result was presented in [31] without a a formal proof.

6.6 Real-time Model of BGP

Despite of the terrifying theoretical bounds on the speed of convergence, BGP continues to be practically the only inter-domain routing protocol used today. Even though users have noticed problems with convergence of BGP, they have not been nowhere nearly as severe as the theory predicts they could be. This section tries to explain this gap. The fundamental question which we address in this section is "how long does BGP take to converge?".

Recall the definition of the oscillation index. The definition does not include any real-time constraints. It assumes that routers can be activated in any order and that they can process routes arbitrarily quickly or slowly. In particular, in the worst-case, routers will react to every single advertisement from any of their neighbors by immediately propagating that advertisement further if it constitutes a more desirable route. For instance, if a router receives ten route advertisements from its neighbors within a short interval of time and if the routes are in the increasing order of preference, that router will propagate *each* of these ten routes to *each* of its neighbors. In reality, however, routers do not advertise immediately. They wait for a certain period of time during which they collect the advertisements, and then they propagate only the best of the newly learned routes. [9] The oscillation index can not capture this consolidation of advertisements, because it does not include any real-time information. It is indeed possible, but not very likely, that a router receives only one new route in every interval during which it collects advertisements. In that case, the oscillation index of that router will be proportional to its convergence time. Therefore, the oscillation index really captures the worst-case convergence if we do not have any information

[9] The routers have to wait for some time, since only one advertisement per destination can be sent be sent within any **MinRouteAdvertisementInterval** seconds.

about the frequency of advertisements. However, including this real-time information enables us to perform a more precise analysis by trimming down the set of possible scenarios. As a result, we will get a tighter bound on the speed of convergence.

Definition: [Timed Stable Paths Problem] We say that a triple $S = ((V, E, d), \mathcal{P}, \Lambda)$ is an instance of the *Timed Stable Paths Problem* (TSPP) if $((V, E), \mathcal{P}, \Lambda)$ is an instance of SPP and $d : V \times V \rightharpoonup \mathbf{R}$ (called *delay*) is a strictly positive partial function such that $d(u, v)$ is defined if and only if $\{u, v\} \in E$. For the sake of notational convenience, we will denote the corresponding SPP instance of S by SPP(S) (i.e. SPP(S) $= ((V, E), \mathcal{P}, \Lambda)$). □

In other words, a TSPP instance is simply an SPP instance where each link has two "delays" associated with it (one for each direction). The purpose of the extension is not to to introduce a new kind of a problem, but rather to constrain the SPVP algorithm. The intuition behind the delay function is the following: for a given edge $\{u, v\} \in E$, $d(u, v)$ is an upper bound on the delay time of route propagation from node u to node v. In other words, if u changes its route at time t, v will find out about that change before or at time $t + d(u, v)$. Notice that establishing any upper bound on convergence time requires explicit or implicit bounds on edge delays. In the absence of even one edge delay bound, a router could ignore the advertisements from its neighbor (and thus postpone the convergence) indefinitely. Obviously, no upper bound on global convergence time could be established in that case.

In order to formalize the intuition of edge delays, we will generalize the notion of an activation sequence introduced earlier. The generalized activation sequence will be a stream of events together with their occurrence times, which we call *time stamps*. In order to perform real-time analysis, we need to allow advertisements to be consolidated by not insisting that a router reacts to every single advertisement. A router u can now wait for some time (less than $d(u, v)$) before notifying its neighbor v of a route change. We model this possibility of delay by separating the event of receiving an advertisement from the event of recomputing and advertising the best route. Effectively, the event *receive UPDATE(...) from p* in Appendix A.4 simply splits into two events:

receive UPDATE $(dest, path)$ *from* p

{

  $ribIn_{p,dest} \leftarrow path$

}

recompute $(dest)$

{

```
    b ← best(dest)
    if (rib_dest ≠ b)
    {
      rib_dest ← b
      for each v ∈ peers do
      {
        send ([UPDATE(dest, rib_dest)], v)
      }
    }
  }
```

The first event denotes a receipt of a route, which is then stored in the appropriate rib_in. The second event recomputes the best route and sends it to the neighbors in case it changed. The above pseudo-code describes how each individual router handles the events. We will now describe precisely how each of the events affects the global system state.

We will not include the destination parameter, since we assume it is always node 0. Therefore, we will denote the events as receive(v $\Leftarrow$ u) and recompute(u). They change the global state in the following way:

- **receive**($\mathbf{v} \Leftarrow \mathbf{u}$):
  rib_in($v \Leftarrow u$) := dequeue (mq($v \Leftarrow u$)).

- **recompute**($\mathbf{u}$):
  if rib(u) $\neq$ best(u) then
  {
    rib(u) := best(u)
    $\forall v \in$ peers(u). enqueue(mq($v \Leftarrow u$), rib(u))
  }

Function *dequeue* removes the front element of the queue and returns it as the result. It signals an error if the queue is empty. Function *enqueue* adds an element at the back of the queue.

Similarly as before, if e is any of the above events, we use the notation $(\mathcal{S}, \mathcal{Q}) \stackrel{e}{\to} (\mathcal{S}', \mathcal{Q}')$ to indicate that e transforms the state $(\mathcal{S}, \mathcal{Q})$ into the state $(\mathcal{S}', \mathcal{Q}')$.

Definition: [Timed activation sequence] *Timed activation sequence* is any finite or infinite sequence of pairs $(e_i, t_i)_{i=1,2,\ldots}$ where each e_i is either of the form recompute(u) or of the form receive(v $\Leftarrow$ u) for some peers u and v, and $t_i < t_j$ for every $i < j$. We say that t_i is a *time stamp* (or time of

occurrence) of the event e_i. We say that a timed activation sequence $(e_i, t_i)_i$ is *initialized* if $t_1 = 0$ and $e_1 = \text{recompute}(0)$. □

When a timed activation sequence $(e_i, t_i)_i$ is run on an initial state $(\mathcal{S}_0, \mathcal{Q}_0)$, it generates a sequence of states states $(\mathcal{S}_i, \mathcal{Q}_i)_{i=0,1,\ldots}$, where

$$(\mathcal{S}_{i-1}, \mathcal{Q}_{i-1}) \xrightarrow{e_i} (\mathcal{S}_i, \mathcal{Q}_i).$$

We say that $(\mathcal{S}_i, \mathcal{Q}_i)$ is the state after time t_i.

Initialized timed activation sequences will be used to describe the evolution of the system starting from the clean initial state. Recall that this is the state defined by:

$$\text{rib}(u) = \begin{cases} \epsilon, & u \neq 0 \\ (0), & u = 0 \end{cases},$$

$\text{rib_in}(u \Leftarrow v) = \epsilon$, and $\text{mq}(u \Leftarrow v)$ contains only $\text{rib}(v)$.

We want to consider only those timed activation sequences that respect the delay function. This is captured by the notion of *validity*, which formalized in the following definition:

Definition: [Validity] A timed activation sequence $(e_i, t_i)_i$ is *valid* with respect to a TSPP instance $S = ((V, E, d), \mathcal{P}, \Lambda)$ and an initial state $(\mathcal{S}_0, \mathcal{Q}_0)$ if the following two conditions hold for every pair of peers u and v when we run the sequence on S initially in the state $(\mathcal{S}_0, \mathcal{Q}_0)$:

1. An event of the form $\text{receive}(v \Leftarrow u)$ never happens when $\text{mq}(v \Leftarrow u)$ is empty.
2. For every i, if $e_i = \text{recompute}(u)$ is an event that causes a route P to be advertised (i.e. $P = \text{best}(u) \neq \text{rib}(u)$), then there exist j and k with the following properties:
   (a) $e_j = \text{receive}(v \Leftarrow u)$ is an event that puts P into $\text{rib_in}(v \Leftarrow u)$.
   (b) $e_k = \text{recompute}(v)$.
   (c) $t_i < t_j < t_k \leq t_i + d(u, v)$.

□

At the first glance, the definition may look complicated, but it has a very natural intuitive meaning. It simply requires that the time interval between u advertising a route and v consuming it must be bounded by $d(u, v)$. This lag interval is composed of two delays:

1. The time that a route advertisement takes to traverse the link (u, v).
2. The time that the received advertisement spends in $\text{rib_in}(v \Leftarrow u)$ before it gets "consumed" (i.e. used by v for recomputing the route).

In the definition, the first delay is $t_j - t_i$ and the second delay is $t_k - t_j$. The definition requires that the sum of these two delays be less or equal $d(u, v)$.

We say that a valid timed activation sequence is *finite* if it has a finite number of "receive" events. If this is the case, then the corresponding sequence of generated states stabilizes. It is easy to see why. After all "receive" events happen, all the communication queues (mq) must become and remain empty, since the second condition from the definition of validity requires that everything put on a queue be eventually received through a "receive" event. But this means that the subsequent "recompute" events would not change the state (if they would, the queues would not remain empty, so there would have to be more "receive" events).

More generally, finite valid activation sequences correspond exactly to convergent behaviors, while non-finite ones correspond to divergent behaviors.

Definition: [Convergence time] We say that a TSPP instance S *converges in time* t from the state $(\mathcal{S}_0, \mathcal{Q}_0)$ if every valid initialized timed activation sequence with respect to S and $(\mathcal{S}_0, \mathcal{Q}_0)$ satisfies the following two conditions:

1. It is finite.
2. It does not change any rib(u) after time τ. □

The definition simply requires that all initialized valid behaviors are convergent (finite) and that routers always find stable paths at or before time t. When proving the first part of this statement, we will sometimes find it convenient to use the notion of *boundedness*.

Definition: [Boundness] Given a TSPP instance S and an initial state $(\mathcal{S}_0, \mathcal{Q}_0)$, we say that a timed activation sequence $(e_i, t_i)_i$ is *bounded* by c if the following statement holds:

$$\forall i, u, v.\ e_i = \text{receive}(u \Leftarrow v) \ \Rightarrow \ t_i \leq c.$$

□

In other words, boundness by c requires that all 'receive' events happen at or before time c. Compare this to the definition of finiteness, which requires that there are finitely many 'receive' events. Obviously, every finite timed activation sequence is also bounded by some constant. One may be tempted to conclude that the reverse is true as well. This is not the case, since *every* valid timed activation sequence $(e_i, t_i)_{i=1,2,\ldots}$ can be made bounded by changing the time stamps. For instance, one way to do this is to define the new time stamps as

$$t'_i := t_1 + (1 - \frac{1}{2^{i-1}})\epsilon$$

for a small ϵ. It is easy to see that one can always find a small enough ϵ so that the resulting timed activation sequence is still valid. Furthermore, the new sequence is obviously still non-finite and bounded by $t_1 + \epsilon$.

Even though boundness does not imply finiteness for each individual sequence, this implication will hold if we universally quantify these properties over all valid initialized timed activation sequences. This is stated in the following lemma.

Lemma 12 *Let S be a TSPP instance and $(\mathcal{S}_0, \mathcal{Q}_0)$ an arbitrary state for S. If all valid initialized timed activation sequences for S and $(\mathcal{S}_0, \mathcal{Q}_0)$ are bounded by some constant c, then they are all finite.*

Proof: Assume, on the contrary, that there is at least one non-finite valid initialized timed activation sequence. For an arbitrary such sequence $T = (e_i, t_i)_{i=1,2,\ldots}$, we can define the limit of T as

$$L(T) := \lim_{i \to \infty} t_i.$$

Because of the boundedness and the fact that T has an infinite number of receive events, we know that $L(T) \leq c$. We will show that there exists a constant μ such that for every non-finite valid initialized timed activation sequence T, we can find another such sequence T' with the property $L(T') = L(T) + \mu$. By repeating this procedure on T' and subsequent sequences, we would eventually obtain a sequence $\overline{T}$ with the property $L(\overline{T}) > c$, which contradicts the boundness assumption from the lemma. This shows that there can be no non-finite valid initialized timed activation sequences.

Let $m = \min\{d(u,v) \mid \{u,v\} \in E\}$ be the smallest edge delay. We claim that we can take $\mu = m/2$. Let $T = (e_i, t_i)_i$ be an arbitrary non-finite valid initialized timed activation sequence. Since $L(T) < c$, there must exist an index i_0 so that

$$\forall i, j.\ i_0 \leq i < j \ \Rightarrow \ t_j - t_i < m/2.$$

Intuitively, after t_{i_0} events happen "too fast" and allow for at least another $m/2$ units of time slack. Therefore, we can postpone them by $m/2$ time units and still have a valid sequence. However, at time t_{i_0} there may still be pending messages which have already been delayed enough and do not have so much of time slack, so we need to wait till they are processed to postpone the subsequent events. Formally, for each pair of peers (u, v), consider the most recent message $m_{u,v}$ in the queue $\mathrm{mq}(v \Leftarrow u)$ at time t_{i_0}. For each such message, let $t_{j_{u,v}}$ be the time when the event $\mathrm{receive}(v \Leftarrow u)$ caused $m_{u,v}$ to be received and let $t_{k_{u,v}} > t_{j_{u,v}}$ be the time when the $\mathrm{recompute}(v)$ event happened so that the second condition for validity holds:

$$t_{i_0} < t_{j_{u,v}} < t_{k_{u,v}} < t_{i_0} + d(u,v).$$

Define $j_0 := \max\{k_{u,v} \mid \{u,v\} \in E\}$. Informally speaking, t_{j_0} is the earliest time when all messages generated before time t_{i_0} have been processed. Now define the new sequence of time stamps in the following way:

$$t'_i := \begin{cases} t_i & \text{if } i \leq j_0 \\ t_i + m/2 & \text{if} i > j_0 \end{cases}$$

It is easy to see that $T' = (e_i, t'_i)_i$ is a valid initialized timed activation sequence such that $L(T') = L(T) + m/2$. □

Just as we have extended the notions of SPP instances and activation sequences with real-time information, we can do the same with dispute digraphs.

Definition: [Timed dispute digraph] Given a TSPP instance $S = ((V,E,d),\mathcal{P},\Lambda)$, the corresponding *timed dispute digraph* $\mathcal{TDD}(S)$ is a weighted directed graph which is structurally identical to $\mathcal{DD}(\text{SPP}(S))$ with the weight function Δ defined in the following way: an edge from $P = (u\ldots)$ to $Q = (v\ldots)$ in the dispute digraph has the weight of $\Delta(P,Q) = d(u,v)$. □

A timed dispute digraph is therefore a weighted dispute digraph, where weights correspond to edge delays. Nodes u and v are neighbors and it is therefore possible that u adopts or rejects the path P at some time t_P, which causes v to adopt or reject the path Q at some later time t_Q. Intuitively, $\Delta(P,Q)$ is meant to represent a bound on the time between t_P and t_Q. Recall, however, that we have introduced $d(u,v)$ with the intention of representing exactly this bound. This is why we defined $\Delta(P,Q) = d(u,v)$. The following key lemma formally describes the fundamental connection between the timed dispute graph and timing of the events.

Lemma 13 (Real-time correspondence) *If $S = ((V,E,d),\mathcal{P},\Lambda)$ is a TSPP instance which is initially in an arbitrary state and $(e_i,t_i)_i$ is a valid initialized timed activation sequence executing on S, then the following holds for any permitted path P: If a node goes up* [10] *to P or goes down* [11] *from P at time t, then there exists a path P^+ with the following properties:*

1. *$P \leq P^+$.*
2. *There is a path in $\mathcal{TDD}(S)$ from (0) to P^+ of the length at least t.*

Proof: Since nodes can change their paths only at times t_i, we can prove the lemma by induction on i. Formally, we will prove the following statement by induction on i:

$$\forall i\ \forall u\ \forall P.\ u \text{ goes (up to)/(down from) } P \text{ at time } t_i \Longrightarrow \exists\ P^+ \ldots$$

[10] *We say that a node* goes up to *P if it switches to P from some less preferred path Q.*
[11] *We say that a node* goes down from *P if it switches from P to some less preferred path Q.*

Assume first $i = 1$. Since the activation sequence is initialized, we know that $t_1 = 0$ and $e_1 = \text{recompute}(0)$. Therefore, only node 0 can be active at this time and P must be (0). But then we can take the empty path starting at (0) as a path of length $t_1 = 0$ from (0) to P.

Now assume that the statement holds for all $i < k$, where $k > 1$. Let u be the node that goes up to P or down from P at time t_k. Since $t_k > 0$ (because $k > 1$), we know that $u \neq 0$, because node 0 does not change its path after the initial moment. This means that $P = (u\ v\ \ldots)$ for some neighbor v of u. There are two possible cases:

Case 1: u goes up to P at time t_k.
Then, because of the definition of valid activation sequences, we conclude that v must have advertised the path $P[v \ldots 0]$ at some time $t' \geq t_k - d(v,u)$ (the delay from v advertising the route till u adopting it is bounded by $d(v,u)$). There are two subcases:

- Case 1a: v went up to $P[v \ldots 0]$ at time t'.
  Since $t' < t_k$, this means that $t' = t_i$ for some $i < k$. By the induction hypothesis, we know that some path $R \geq P[v \ldots 0]$ is reachable in $\mathcal{TDD}(S)$ from (0) by a path of the length at least t'. There are two possible subcases with respect to u's preference:

  - Case $P \leq (u\ v)R$. Since there is a transmission arc from R to $(u\ v)R$ with the weight $d(v,u)$, we conclude that $(u\ v)R$ is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ := (u\ v)R$. Notice that $P \leq P^+$ because of the assumption of this subcase.
  - Case $(u\ v)R < P$. In this case we have a dispute arc from R to P with the weight $d(v,u)$, so we conclude that P is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ = P$.

- Case 1b: v went down to $P[v \ldots 0]$ (from some path Q) at time t'. As in case 1a, we conclude that $t' = t_i$ for some $i < k$. By the induction hypothesis, we know that there exists a path $Q^+ \geq Q$, which is in $\mathcal{TDD}(S)$ reachable from (0) by a path of the length at least t'. Also, since v went down from Q, we know that

  $$P[v \ldots 0] \leq Q \leq Q^+.$$

  As before, there are two possible subcases with respect to u's preference:

  - Case $P \leq (u\ v)Q^+$. Since there is a transmission arc from Q^+ to $(u\ v)Q^+$ with the weight $d(v,u)$, we conclude that $(u\ v)Q^+$ is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ := (u\ v)Q^+$.

– Case $(u\ v)Q^+ < P$. In this case, we have a dispute arc from Q^+ to P with the weight $d(v,u)$, so we conclude that P is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ := P$.

Case 2: u goes down from P at time t_k.

Then it must be the case that v switched from the route $P[v \ldots 0]$ to some other route at some time $t' \geq t_k - d(v,u)$. Indeed, if v had held $P[v \ldots 0]$ throughout the interval $[t_k - d(v,u), t_k]$, then u would not have gone down from P. Also, if v had held routes different from $P[v \ldots 0]$ throughout the whole interval, u could not have had P just before the time t_k (u would have learned some of those other routes, since the delay time is bounded by $d(v,u)$).

There are two cases with respect to the direction in which v moved:

- Case 2a: v went up to some path Q from $P[v \ldots 0]$ at time t'. As before, we know that $t' = t_i$ for some $i < k$. By the induction hypothesis, we know that some path $Q^+ \geq Q$ is reachable from (0) by a path of the length at least t'. Also, since v went up to Q, we know that

$$P[v \ldots 0] \leq Q \leq Q^+.$$

  There are two possible subcases with respect to u's preference:

  – Case $P \leq (u\ v)Q^+$. Since there is a transmission arc from Q^+ to $(u\ v)Q^+$ with the weight $d(v,u)$, we conclude that $(u\ v)Q^+$ is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ := (u\ v)Q^+$.

  – Case $(u\ v)Q^+ < P$. In this case, we have a dispute arc from Q^+ to P with the weight $d(v,u)$, so we conclude that P is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ := P$.

- Case 2b: v went down from $P[v \ldots 0]$ at time t'. As before, we know that $t' = t_i$ for some $i < k$. By induction hypothesis, we know that some path $R \geq P[v \ldots 0]$ is reachable in $\mathcal{TDD}(S)$ from (0) by a path of the length at least t'. There are two possible cases with respect to u's preference:

  – Case $P \leq (u\ v)R$. Since there is a transmission arc from R to $(u\ v)R$ with the weight $d(v,u)$, we conclude that $(u\ v)R$ is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ := (u\ v)R$.

  – Case $(u\ v)R < P$. In this case, we have a dispute arc from R to P with the weight $d(v,u)$, so we conclude that P is reachable by a path of the length at least $t' + d(v,u) \geq t_k$. Therefore, we can take $P^+ := P$.

□

Notice that cases 1a and 2b have the same proof, as well as 1b and 2a. This means that the overall proof could be "compressed". Nevertheless, the proof is presented in the original form for the sake of clarity.

The real-time correspondence lemma describes a connection between paths of the timed dispute digraph and timing of events. A simple corollary of the lemma would enable us to estimate convergence time for some TSPP instances. Before we present this result, it is convenient to introduce one more notion:

Definition: [Diameter] Let $G = \mathcal{TDD}(S)$ be the timed dispute digraph of a TSPP instance S. The *diameter* of G (shortly $D(G)$) is the length of the longest path in $\mathcal{TDD}(S)$ which starts at the node (0). □

This notion seems slightly different from the standard notion of diameter, which simply denotes the length of the longest path in a weighted graph, regardless of where it starts or ends. Nevertheless, in a timed dispute digraph, there is always a path from (0) to any other node (otherwise, the node can be ignored [12]), so the longest path in the digraph will necessarily start at (0). Notice also that, since all edges have positive weights, a timed dispute digraph will be acyclic if and only if it has a finite diameter.

The following theorem gives an upper bound on convergence time of TSPP instances with acyclic dispute digraphs.

Theorem 6 (Upper bound on SPVP convergence time) *Let S be a TSPP instance such that $\mathcal{TDD}(S)$ is acyclic with the diameter $D(\mathcal{TDD}(S)) = \tau$. Then S converges in time τ from any initial state.*

Proof: We need to show two things:

1. All valid initialized timed activation sequences are finite.

2. No valid initialized timed activation sequence changes any $\text{rib}(u)$ field after the time τ.

Let us first prove the second statement, since it follows directly from the real-time correspondence lemma.

Assume, on the contrary, that for some initial state $(\mathcal{S}_0, \mathcal{Q}_0)$ and some valid initialized timed activation sequence $(e_i, t_i)_i$, some router u changes $\text{rib}(u)$ at some time $t > \tau$. This means that u

[12] For each permitted path P, the dispute graph should contain a path from (0) to P composed solely of transmission arcs. If this is not the case, P can never be learned through advertisements initiated at the destination 0, so P is a vacuous route that can be deleted from the dispute digraph for the purpose of convergence analysis.

either goes up to some path P or down from some path P at the time t. But Lemma 13 in that case guarantees that there exists a path in $\mathcal{TDD}(S)$ from (0) to some node P^+ of the length at least as big as t. However, since $t > \tau$, this contradicts the assumption that $\mathcal{TDD}(S)$ has the diameter τ. This shows the second statement.

For the first statement we will first use Lemma 12. Because of the lemma, it suffices to show that all valid initialized timed activation sequences are bounded by some constant c. Let $M = \max\{d(u,v) \mid \{u,v\} \in E\}$ be the maximum edge delay. We will prove that $c = \tau + M$ is a good bound.

Assume, on the contrary, that this is not the case—that is, there exists a valid initialized timed activation sequence $(e_i, t_i)_i$ such that the following holds

$$\exists j, u, v.\ e_j = \text{receive}(v \Leftarrow u) \ \wedge \ t_j > \tau + M.$$

In other words, v receives a route from u at some time later than $\tau + M$, and in particular, later than $\tau + d(u,v)$. Because of validity, this means that u must have sent that route at some time $t > \tau$. Therefore, u must have gone up to some path P or down from some path P at the time t. However, this is not possible because of the exact same argument used in the proof of the second statement. $\square$

6.6.1 Consistency of Real-time Constraints

Recall the role of the MinRouteAdvertisementInterval parameter from the BGP standard: it is the minimum separation interval of time between two consecutive route advertisements for the same destination. This constraint prevents routers from advertising too frequently. On the other hand, notice that edge delays in TSPP prevent routers from advertising too infrequently. Therefore, these two types of real-time constraints are in some sense duals of each other. Because of that, it is theoretically possible that they contradict each other in the sense that a router can not respect both of them at the same time. We would like to provide some sanity conditions on these parameters that would prevent such inconsistencies.

For the neighboring routers u and v, let $m(u,v)$ denote the value of the MinRouteAdvertisementInterval for route advertisements from u to v. When some other neighbor w sends an advertisement to u, the advertisement goes through two phases before it is consumed by u:

1. Traversing the link (w, u).

2. Waiting at u to be processed.

Let $l(w,u)$ denote the time taken by the first phase. Since the total delay of that advertisement should be bounded by $d(w,u)$, the second phase should take no longer than $d(w,u)-l(w,u)$ of time. At the end of the second phase, the route needs to possibly be disseminated to other neighbors. This means that in the worst case, u can be required to advertise every $d(w,u)-l(w,u)$ units of time. However, in order to allow enough time for proper spacing of consecutive advertisements dictated by MinRouteAdvertisementInterval, this time interval should be at least as big as $m(u,v)$ for every neighbor v. Therefore, we get the following inequality for u:

$$\min_w (d(w,u) - l(w,u)) \geq \max_v m(u,v). \tag{6.1}$$

Minimum and maximum are taken over the set of neighbors of u. If the inequality 6.1 is violated by some w and v, we can easily construct a scenario where u would theoretically not be able to respect the edge delay constraint for w without flooding v with overly frequent advertisements.

The inequality can also help us determine sensible edge delays for estimating convergence time. Suppose that all $m(u,v)$ are given—in most cases they are set to the recommended value of 30 seconds. Let $M(u) = \max_v m(u,v)$ be the maximum outgoing MinRouteAdvertisementInterval. The inequality 6.1 will be satisfied if we choose

$$d(w,u) \geq M(u) + l(w,u),$$

for every neighbor w. The value of $l(w,u)$ depends on the quality of the TCP connection between w and u. In practice, however, this value will be several orders of magnitude smaller than $M(u)$. By making this practically safe assumption, we arrive at the following estimate for edge delays:

$$d(w,u) \geq 2M(u).$$

The estimate is relatively crude, since, for instance, it assigns the same edge delay to all the incoming edges for u. Nevertheless, it suffices for satisfaction of the inequality 6.1 in most practical situations.

6.7 Application: Delay-cost Consistent Policies

One of the ways we can avoid inconsistent routing policies is to have the routers agree on a common *cost function* which is used to assign route preferences. This idea is presented in [19]. For a given TSPP (or an SPP) instance, a cost function is a strictly positive partial function $c : V \times V \rightharpoonup \mathbf{R}$ such that $c(u,v)$ is defined if and only if $\{u,v\} \in E$. [13] Any cost function naturally extends to

[13] The requirement for strict positivity can be weakened into the requirement that c does not result in any non-positive cycles in the underlying graph. See [19] for details.

paths—if $P = (u_1\ u_2\ \ldots\ u_k)$ is a path, then we define

$$c(P) := \sum_{i=1}^{k-1} c(u_i\ u_{i+1}).$$

We say that S is *consistent* with the cost function c (shortly, *cost-consistent*) if the following holds for every node u:

$$\forall P, Q \in \mathcal{P}^u.\ c(P) < c(Q) \ \Rightarrow\ \lambda^u(Q) < \lambda^u(P).$$

For TSPP instances, we can also extend the delay function to paths. For $P = (u_1\ u_2\ \ldots\ u_k)$, we define

$$d(P) := \sum_{i=k}^{2} d(u_i\ u_{i-1}).$$

Notice that the summation goes in the opposite direction from the one used in the cost formula. The intuition behind this is that delays bound the propagation time of advertisements that go from the destination *outwards*, while cost is supposed to measure how expensive is to send the traffic along the route *towards* the destination.

It is natural to expect that delays will be reflected in the cost function in the sense that routes with longer delays should be considered more expensive for carrying traffic and vice-versa. This is captured by the following definition.

Definition: [Delay-cost consistency] We say that a TSPP instance S consistent with a cost function c is *delay-cost consistent* if for every node u the following condition holds:

$$\forall P, Q \in \mathcal{P}^u.\ d(P) < d(Q) \ \Rightarrow\ c(P) < c(Q).$$

□

One way to achieve delay-cost consistency is, for instance, by assigning costs according to the following rule:

$$c(u, v) = f(d(v, u)),$$

for some strictly increasing positive function f.

The following lemma establishes an upper bound on the length of a path in the timed dispute digraph of a delay-cost consistent instance. The bound is interesting in the sense that it depends only on the last node of the path.

Lemma 14 *Let S be a delay-cost consistent instance with the cost function c. If $\Pi = (0) \to P_1 \to \ldots \to P_k$ is a path in $\mathcal{TDD}(S)$, then its length is less or equal $d(P_k)$.*

Proof: Let $L(.)$ denote the length function on paths in $\mathcal{TDD}(S)$. We will prove the statement by induction on k.

If $k = 0$, we have a trivial path $\Pi = (0)$ which does not have edges and hence $L(\Pi) = 0 = d((0))$.

Assume the statement is true for every path in $\mathcal{TDD}(S)$ with less than m edges, for some $m \geq 1$. Let $\Pi = \Pi' \to P_m$ be a path in $\mathcal{TDD}(S)$ with m edges, where $\Pi' = (0) \to P_1 \to \ldots \to P_{m-1}$ is its sub-path consisting of the first $m-1$ edges. Let us denote the first node of P_i by u_i (i.e. $P_i = (u_i \ldots)$). Then by the induction hypothesis, we know that

$$L(\Pi) = L(\Pi') + \Delta(P_{m-1}, P_m) \leq d(P_{m-1}) + d(u_{m-1}, u_m). \tag{6.2}$$

There are two possible cases with respect to the nature of the arc $P_{m-1} \to P_m$:

1. $P_{m-1} \to P_m$ is a transmission arc. Then $P_m = (u_m\ u_{m-1})P_{m-1}$. By the definition of path delays we have
$$d(P_m) = d(P_{m-1}) + d(u_{m-1}, u_m).$$
This is exactly the right-hand side of the equation 6.2, which means that $L(\Pi) \leq d(P_m)$.

2. $P_{m-1} \to P_m$ is a dispute arc. Then $P_m = (u_m\ u_{m-1})Q_{m-1}$, where $Q_{m-1} \leq P_{m-1}$. Because of the cost consistency, we know that $c(P_{m-1}) \leq c(Q_{m-1})$. Delay-consistency then implies that $d(P_{m-1}) \leq d(Q_{m-1})$. If we plug this into the right-hand side of the equation 6.2, we get:
$$L(\Pi) \leq d(Q_{m-1}) + d(u_{m-1}, u_m) = L(P_m).$$

□

We can now derive an upper bound on convergence time:

Corollary 1 (Convergence time of delay-cost consistent instances) *Let S be a delay-cost consistent TSPP instance with the maximum path delay*
$\delta = \max\{d(P) \mid P \in \mathcal{P}^u,\ u \in V\}$. Then S converges in time δ.

Proof: Because of Lemma 14, the diameter of $\mathcal{TDD}(S)$ is bounded by δ. But then Theorem 6 guarantees that S converges in time δ. □

BGP routers are often configured to simply prefer routes with smaller number of hops. Formally, we say that an SPP instance has *shortest-path-first* policies if it is consistent with the cost function that assigns unit costs to all edges:

$$c(u, v) = 1, \quad \text{for } \{u, v\} \in E.$$

The following theorem estimates convergence time for instances with shortest-path-first policies:

Theorem 7 (Convergence time for shortest-path-first policies) *Let S be a TSPP instance with shortest-path-first policies, where all edge delays are equal to ω.* [14] *Then S converges in time $D\omega$, where D is the length of the longest permitted path.*

Proof: First notice that having all edge delays be equal to ω yields a delay-cost consistent instance, since $c(u, v) = \frac{1}{\omega}d(v, u)$ and consequently,

$$c(P) = \frac{1}{\omega}d(P).$$

The delay of a path P is equal to $|P| \cdot \omega$, where $|P|$ is the length of P. Therefore, the maximum path delay will be $D\omega$, which by Corollary 1 implies that S converges in time $D\omega$. □

This theorem is due to Labovitz et al. [34], but here we showed how to derive it as a corollary of the real-time correspondence lemma.

6.8 Application: Convergence of SPVP

Another example that illustrates the power of the real-time correspondence lemma is the main convergence result for the "timeless" version of SPVP proved by Griffin, Shepherd and Wilfong in [19]:

Theorem 8 (Sufficient condition for convergence of SPVP) *If S is an SPP instance whose dispute digraph is acyclic, then SPVP is guaranteed to converge on S under any fair activation sequence.*

Proof: Theorem 6 is essentially a refinement of this theorem. We will use it to prove this "timeless" version by showing that the timed protocol can simulate any scenario from the timeless protocol by simply adding some timing information in a consistent way.

Suppose we are given a fair activation sequence $(r_i)_{i=1,2,...}$ for S. We first construct the corresponding TSPP instance S' from S by simply making all edge delays equal to, say, 1. Then we use the activation sequence for S to construct an equivalent valid initialized timed activation sequence $(e_i, t_i)_{i=1,2,...}$ for S' by adding time stamps. We do that in the following way: if the i-th untimed event constitutes of activating the router r_i which processes a message from the queue $\mathrm{mq}(r_i \Leftarrow p)$, then the corresponding timed events will be

$$e_{2i} = \mathrm{receive}(r_i \Leftarrow p),\ e_{2i+1} = \mathrm{recompute}(r_i).$$

[14] *If edge delays are different, we can take ω to be the largest edge delay.*

The difference in indices comes from the fact that each untimed event corresponds to two timed events. The initial event is $e_1 = \text{recompute}(0)$. Finally, the time stamps are defined as

$$t_i = 1 - \frac{1}{2^{i-1}}.$$

Since $t_j - t_i < 1$ for any $j > i$, we know that all edge delays will be respected, so that $(e_i, t_i)_i$ really constitutes a *valid* timed activation sequence. Also, since $t_1 = 0$ and $e_i = \text{recompute}(0)$, the sequence will be initialized. Because of the assumed acyclicity of the dispute digraph, we can use Theorem 6 which guarantees that the sequence $(e_i, t_i)_i$ converges on S'. Moreover, it is easy to see (from the pseudo-code) that the state of S after activating r_i will be the same as the state of S' at time t_{2i+1}. The observation allows us to conclude that the original activation sequence $(r_i)_i$ converges on S. □

6.9 Application: Safe SPVP

We have seen that BGP currently does not ensure convergence. One would like to fix this problem without sacrificing much of the freedom that BGP provides in terms of the choice of routing policies. We would like to minimally constrain the protocol in a way that would guarantee convergence.

One proposal for doing this is described by Griffin and Wilfong in [21]. This is a dynamic solution that modifies the policies "on-the-fly" (i.e. while the protocol is running). In addition to paths, routers also propagate *history traces*, which are used to detect potentially divergent behaviors. When a possibility of divergence is detected, the policies are modified in a way that avoids problematic paths. Despite the elegance of the idea, this solution has two practical drawbacks:

- Carrying and remembering history traces can use a significant amount of system resources—primarily bandwidth and memory.
- The solution requires substantial modification of BGP implementations.

A quite different approach is presented by Gao and Rexford in [17]. Their approach is static. Instead of regarding the internetwork as a flat structure where all routers are considered equal, they use the provider-customer hierarchy of the Internet to statically configure BGP in a way that guarantees convergence. Precisely, they assume that *every* BGP peering edge belongs to exactly one of the following two classes:

Customer-provider edges exist between a *customer* and a *provider*. The provider is typically a larger AS that provides connectivity to the rest of the Internet for the customer. Providers may have even larger providers of their own and customers may have further customers.

Peer-to-peer edges [15] exist between two AS's of a comparable size, who mutually agree to exchange traffic between their respective *customers*. In particular, this means that peer-to-peer edges can only be included in routes that go from a (direct or indirect) customer of one peer to a (direct or indirect) customer of the other peer.

An instance of SPP which has this hierarchy is called a *hierarchical instance* of SPP. We should emphasize the fact that this hierarchy already exists on the Internet. Therefore, the hierarchy is an observation, rather than an assumption. We will view the customer-provider edges as directed edges, oriented from the provider to the customer. In that sense, they form a directed graph which we always assume to be *acyclic*. Peer-to-peer edges are naturally undirected, since the peering relationship is symmetric. Figure 6.5 shows an example of this hierarchy. Customer-provider edges are represented with full lines, while peering edges are dashed.

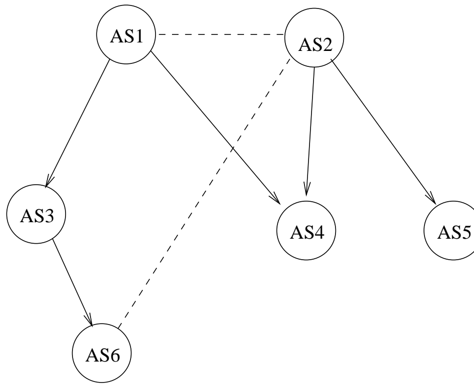

Figure 6.5: Hierarchical topology

Without the loss of generality, we can assume that we have a single BGP router for every AS. If r is a router, then provider(r), customer(r), and peer(r) respectively denote the sets of (immediate) *providers*, *customers* and *peers* of r. Let $P = (r\ r_1\ r_2\ \ldots\ r_k)$ be a route that r learns from another AS. We use the notation next(P) to denote the router that would be used as the next hop on P. In this case, next(P) $= r_1$. All of r's routes can be classified into three categories based on the next hop:

Customer routes: routes P, such that next(P) $\in$ customer(r).

Provider routes: routes P, such that next(P) $\in$ provider(r).

Peer routes: routes P, such that next(P) $\in$ peer(r).

[15] In order to avoid confusion, from now on we will use the term *peers* for any two routers connected by a peer-to-peer edge, and the term *BGP peers* for any two routers that run a BGP session between them.

Convergence is assured by restricting two aspects of BGP: flow of advertisements and routing policies. Figure 6.6 shows selective export rules that should be used to restrict the flow of advertisements. The rules specify which routes should be advertised to which neighbors.

Figure 6.6: Selective export rules

		Export to		
		provider	customer	peer
Type	provider route	NO	YES	NO
	customer route	YES	YES	YES
	peer route	NO	YES	NO

The table should be interpreted for any fixed router (i.e. for any router r, the table prescribes the rules for exporting r's provider, customer and peer routes to r's providers, customers and peers). It says that no router should advertise its peer and provider routes to other peers and providers. This is obviously different from the original BGP/SPVP, where all routes are advertised to all BGP peers. Nevertheless, the constraints follow the described hierarchy quite naturally. Let us take a closer look at each of the NO entries in the table.

- Provider routes should not be exported to other providers. A router should use its customer routes to reach its direct and indirect customers. For all other destinations, the router should use a provider or perhaps a peer, if one is available. Exporting a route learned from one provider to another provider would potentially create a route between the two providers that goes through their common customer. This is a very unnatural situation. In that case, one provider would use its customer route to reach a non-customer (the other provider), which violates the intended use of customer routes. Instead of exchanging their traffic through a common customer, the two providers should exchange their traffic through a common (direct or indirect) provider.

- Peer routes should not be exported to providers and vice-versa. Recall that peering edges are supposed to be used for traffic between the peers' respective customers. Exporting a peer route to a provider would potentially create a route that uses the peering edge for the traffic between one peer's *provider* and the other peer's customers. A similar route in the opposite direction can be created when exporting a provider route to a peer.

- Peer routes should not be exported to other peers. Doing so would again violate the intended use of peering edges. It could create a route which uses a peering edge to exchange traffic between one peer and the other peer's *peer*. In effect, this would automatically peer any two

routers which have a common peer. Consequently, it would force any two ISPs that want to become peers into having identical peer sets. Formally speaking, this export rule allows the peering relation to be non-transitive.

Selective export rules can be easily configured on top of the existing BGP. They only require appropriate setting of the BGP *export policies* which determine what gets stored in the Adj-RIB-Out tables. Selective export rules impose a certain shape on the paths that can be produced.

Definition: [Valley-free paths] A path $(r_1\ r_2\ \dots\ r_k)$ is *valley-free* if every provider-to-customer or a peer-to-peer edge can only be followed by a provider-to-customer edge.
Formally, if for some $1 < i < k$, $(r_{i-1}\ r_i)$ is a provider-to-customer or a peer-to-peer edge, then $(r_i\ r_{i+1})$ is a provider-to-customer edge. □

In other words, a valley-free path consists of a sequence of zero or more customer-to-provider edges, followed by zero or one peer-to-peer edge, followed by a sequence of zero or more provider-to-customer edges. Graphically, if we imagine that providers are placed higher than customers, then valley-free paths look like "hills"—they do not allow the path to go "up", once it started going "down", hence the name valley-free. The notion of valley-free paths is due to Lixin Gao and is described in [16].

Theorem 9 (SPVP valley-free paths) *If all routers respect the selective export rules from Figure 6.6, then SPVP produces only valley-free paths.* □

Intuitively, the theorem says that routers will keep forwarding their traffic "upwards" to providers and providers of their providers until the destination appears among the direct or indirect customers of a provider. After that, the traffic may or may not cross one peer-to-peer edge. Finally, the traffic will start going "downwards" to the destination. The theorem is presented and proved in [16].

Selective export rules restrict the shape of the routes that are produced by SPVP. In order to guarantee safety, we need to supplement these rules by guidelines for configuring routing policies. Policy guidelines restrict the set of legal preference functions. It is important that these guidelines are *local*, which means that each router can follow them independently of other routers. The following basic policy guideline is suggested in [17]:

Guideline A: For every router r and every two routes P and Q to the same destination, the following condition holds:

$$(\text{next}(P) \in \text{customer}(r) \quad \wedge \quad \text{next}(Q) \notin \text{customer}(r)) \ \Rightarrow \lambda^r(P) > \lambda^r(Q).$$

□

The guideline simply says that routers should strictly prefer customer over non-customer routes. The routers have a practical incentive to follow this guideline, since they do not have to pay their customers for transit services. Notice, also, that the guideline does not restrict the router's preference among customer routes, or among peer or provider routes. It turns out that the combination of this guideline and the selective export rules guarantees convergence:

Theorem 10 (SPVP safety) *If all routers obey the selective export rules and Guideline A, SPVP is guaranteed to reach a stable state under any fair activation sequence.* □

Gao and Rexford presented and proved this theorem in [17].

This approach to achieving safety has several important features. First of all, rather than using a new protocol, the suggested restrictions work by simply reconfiguring the existing BGP. This contributes significantly to the usefulness of the approach, since BGP is currently by far the most popular inter-domain routing protocol. Secondly, *global* safety is achieved by applying *local* restrictions on every router, without the need for any kind of global coordination. This is important, since routing policies are in many cases proprietary.

Now that convergence is guaranteed, we would like to know how fast it is. Our first attempt to answering this question is by estimating the oscillation index.

Theorem 11 (Oscillation of safe SPVP) *For some constant c and every $n \geq 2$, there exists an n-node hierarchical SPP instance S_n with the following properties:*

1. *S_n obeys the selective export rules from Figure 6.6.*

2. *All the policies satisfy Guideline A.*

3. *$OI(S_n) \geq c(3/2)^n$.*

Proof: For a given $n \geq 2$, let S_n have the topology depicted in Figure 6.7. Assume that AS's are activated in the order $0, 1, 2, \ldots$. Each AS is activated as many times as needed to consume all the queued advertisements from its customers. Similarly as in the proof of Theorem 5, we assume that more recent routes are always preferred over the earlier ones. Each node $k \geq 2$ will receive the advertisements containing all the routes that its customers $k-1$ and $k-2$ learn. Therefore, we will have

$$OI(k, S_n) \geq OI(k-1, S_n) + OI(k-2, S_n),$$

with $O(0, S_n) = O(1, S_n) = 1$. Therefore, $O(n-1, S_n) \geq F_{n-1}$, where F is the sequence of Fibonacci numbers given by the recursion

$$F_0 = F_1 = 1, \quad F_{k+1} = F_k + F_{k-1} \quad \text{for } k \geq 1.$$

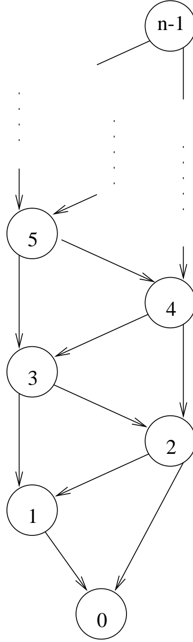

Figure 6.7: A slow converging instance of safe SPVP.

It is known that Fibonacci numbers satisfy the following inequality for some constant c:

$$F_k \geq c(\frac{1+\sqrt{5}}{2})^{n+1}.$$

Therefore, we have

$$OI(S_n) \geq OI(n-1, S_n) \geq F_{n-1} \geq c(\frac{1+\sqrt{5}}{2})^n \geq c(\frac{3}{2})^n.$$

Notice that all the routes in this case are customer routes and they get advertised only to providers, so that selective export rules are satisfied. Also, the policy guideline is trivially satisfied, since the guideline does not restrict the preference among customer routes. □

The result seems to indicate the possibility of a very slow convergence. Moreover, the theorem does not give an upper bound on the oscillation index—rather, it gives a lower estimate of how bad it can be. In fact, upper bound is significantly higher than $O((3/2)^n)$.

However, our real-time model from Section 6.6 provides a more optimistic and a more realistic answer. As the first step, we analyze the shape of the dispute digraph. Since safe SPVP does not allow advertising all routes to all neighbors, dispute digraphs for hierarchical instances will have fewer arcs. Precisely, if selective export rules dictate that the route P should not be exported from v to u, then the corresponding dispute digraph does not contain the transmission arc from P to $(u\ v)P$. Now consider the following observation:

Lemma 15 *Let S be a hierarchical instance of SPP. Let $P = eP'$ and $Q = fQ'$ be permitted paths such that the dispute digraph for S contains a dispute or a transmission arc $P \rightarrow Q$. Then the following holds: If f is a provider-to-customer or a peer-to-peer edge, then e is a provider-to-customer edge.*

Proof: Let $f = (u\ v)$. There are two possible cases, corresponding to the nature of the arc $P \rightarrow Q$:

1. $P \rightarrow Q$ is a transmission arc. Therefore, $Q' = P$. There are three subcases with respect to the edge f:

   - $f = (u\ v)$ is a customer-to-provider edge. We do not have to prove anything in this case.
   - $f = (u\ v)$ is a provider-to-customer edge. Because the transmission arc $P \rightarrow (u\ v)P$ exists, v can advertise P to its provider u. Selective export rules in that case guarantee that P is a customer route, which means that e is a provider-to-customer edge.
   - $f = (u\ v)$ is a peer-to-peer edge. Again, since we have a transmission arc $P \rightarrow (u\ v)P$, v can advertise P to its peer u. But then P must be a customer route, which means that e is a provider-to-customer edge.

2. $P \rightarrow (u\ v)Q'$ is a dispute arc. There are, again, three subcases with respect to the edge f:

   - $f = (u\ v)$ is a customer-to-provider edge. Again, we do not have to prove anything in this case.
   - $f = (u\ v)$ is a provider-to-customer edge. Since v can advertise Q' to its provider u (u would otherwise have no way of learning $Q = (u\ v)Q'$, so we could delete Q from the dispute digraph), Q' must be a customer route. However, since $P \rightarrow (u\ v)Q'$ is a dispute arc, v prefers P over Q'. In that case P must be a customer route as well, since a node can not prefer a non-customer route over a customer route (Guideline A). Therefore, e is a provider-to-customer edge.
   - $f = (u\ v)$ is a provider-to-customer edge. Again, since v can advertise Q' to its peer u, Q' must be a customer route. However, because there is a dispute, v prefers P over Q', so P must be a customer route as well. Therefore, e is a provider-to-customer edge

□

We can inductively extend the lemma to arbitrary paths in the dispute digraph:

Corollary 2 *Let S be a hierarchical instance of SPP. Let $P_1 = e_1 P_1'$, $P_2 = e_2 P_2$, ..., $P_k = e_k P_k'$ be permitted paths, such that the dispute digraph for S contains the path $P_1 \to P_2 \to \ldots \to P_k$. Then the sequence of edges $e_k e_{k-1} \ldots e_1$ represents a valley-free path in S.* □

Let us try to explain the result informally. Given a path in the dispute graph, we can extract the first edges of all the all the routes along that path. These edges, considered in the reverse order, will themselves form a path between some two routers. The corollary says that every path obtained this way will be valley-free. Since a valley-free path contains at most $2n$ edges (where n is the number of routers), every path in the dispute graph must be of the length at most $2n$:

Corollary 3 *If S is a hierarchical instance of SPP, any path in $\mathcal{DD}(S)$ has the length less or equal $2n$.* □

When timing assumptions are added to hierarchical SPP instances, we talk about *hierarchical TSPP instances*. As a direct consequence of the Corollary 3, we derive the following theorem which gives an upper bound on convergence time for hierarchical TSPP instances.

Theorem 12 (Convergence time of safe SPVP) *Let S be a hierarchical TSPP instance with n routers and let $M = \max\{d(u,v) \mid \{u,v\} \in E\}$ be the maximum edge delay. Then safe SPVP converges on S in time $2nM$.*

Proof: Because of Corollary 3, the diameter of $\mathcal{TDD}(S)$ is bounded by $2nM$, so Theorem 6 guarantees convergence in time $2nM$. □

The theorem shows that convergence time of safe SPVP is linear in the number of routers, given that edge delays are bounded by a constant. However, Theorem 11 shows that the oscillation index can be exponential in the number of nodes. Let us take a closer look at the reason for this apparent discrepancy. Consider the example from Figure 6.7. Every AS other than $n-1$ and $n-2$ has two providers that it advertises to. This means that the number of advertisements that an AS sends upstream (i.e. to its providers) can be as big as twice the number of advertisements that the same AS receives from below (i.e. from its customers). Eventually, all those advertisements will reach the top provider, which will experience the largest oscillation. The oscillation index grows exponentially as we move higher upstream. The essential property of this scenario is that every node immediately reacts to *every* advertisement by propagating it upstream. Although this is a theoretically possible behavior, it is not likely to happen in reality. In fact, this worst-case scenario would be impossible with most BGP implementations. In reality, a router waits for a certain period of time during which it may receive multiple advertisements from its customers. It is not before the end of this period that the router finally picks the best route and advertises it to its providers.

This can eliminate a lot of useless control traffic. For instance, if a router first receives a route P_1, and then shortly after that P_2 from the same customer, there is no need to propagate P_1. A somewhat surprising fact is that slowing routers down may actually improve the convergence!

As a conclusion, we can say that both measurements (oscillation index and real-time) are useful. It would be unfair to say that one is more correct than the other. The oscillation index is based on less information, so it allows a richer set of scenarios. Consequently its worst-case scenario will be worse than that of the real-time model. The real-time model should be used in cases where we can estimate bounds on edge delays. These bounds depend on the quality of network links, as well as particular BGP implementations.

6.10 Summary

BGP is the principal inter-domain routing protocol. It allows the users to employ their own route preference policies. It has been shown that inconsistent policies can can cause divergence. We first describe the *Simple Path Vector Protocol (SPVP)*—a formal model of BGP introduced by Griffin et al. We then introduce the *oscillation index (OI)* as the maximum number of route updates that a given router can perform during convergence. We show that even on convergent configurations, the oscillation index can grow factorially with the total number of routers.

Oscillation index does not measure time, but only the number of steps needed for convergence. In order to be able to measure convergence time, we construct a real-time model of the protocol. In the model, every edge has the associated *delay*, which represents the latency of route propagation across that edge. Given a network of routers together with their route preferences, one can construct the *dispute digraph* for that instance of SPVP. It has been shown that acyclicity of the dispute digraph guarantees convergence of SPVP (but not vice-versa). We generalize the notion of dispute digraphs into *timed dispute digraphs*, which can be constructed for *real-time* instances of SPVP. Our main theorem states that convergence time of an SPVP instance is bounded by the diameter of the corresponding timed dispute digraph. We show how to use this general upper bound in three applications.

In the first application we show how to estimate convergence time in cases where route preferences are based on some cost function on edges.

In the second application, we show that our result generalizes the previous theorem about convergence in the case of acyclic dispute digraphhs.

In the last application, we consider *safe SPVP*, which is a restriction of the original protocol, shown to always converge. Safe SPVP represents the current best practices in configuring BGP.

We prove an upper bound on convergence time of safe SPVP which linearly depends on the number of routers and maximum edge delay.

Chapter 7

Conclusions

Routing protocols represent a corner stone of virtually every networking architecture. Entire network communication relies on the routing tables maintained by routing protocols. It is, therefore, essential to assure that these protocols behave correctly. Testing does not entirely suffice for that purpose, since it analyzes only an average-case behavior and hence it can not *guarantee* the absence of errors. Moreover, testing is applied to a particular implementation, not a protocol standard. Because of that, it is important to supplement testing with formal correctness analysis of the protocol *design*. The goal of such analysis is to provide formal guarantees that apply to every possible run of every possible implementation of the protocol. As shown in our AODV case study, formal analysis can uncover errors that can be hardly detected by testing. This especially applies to the cases where errors surface only under some boundary conditions which rarely occur in practice.

Our goal was to study the practice, rather than the theory of correctness analysis of routing protocols. This is why we did not focus on developing new tools, but rather on practical analysis of real routing protocols. The work is organized around three case studies, each dedicated to a different routing protocol. We have analyzed these protocols with respect to the requirements which are of practical concern to the users. Our contributions, which we summarize below, can be divided in two areas: results about protocols and analysis methodology.

Results about protocols:

- RIP: We proved convergence of the protocol and established a sharp upper bound on convergence time.
- AODV: We detected several loop-forming scenarios, suggested modifications and proved that the modified protocol is loop-free. Some of these results have been incorporated in

the next version of the protocol standard.

- BGP: We have established a timeless measure of the worst-case speed of convergence (the *oscillation index*). We refined this measure by developing a real-time model of the protocol, which we used to prove a general bound on convergence time. The power of this general result is illustrated by deriving several useful corollaries about convergence in specific circumstances.

Analysis methodology: We developed a language POPSCL for expressing pseudo-code for routing protocols. We have established new proof techniques for reasoning about routing protocols. One example of this is a general technique of analyzing convergence time of BGP by using *timed dispute digraphs*. Other examples include abstractions, which, we believe, can be re-used to prove other properties of the same protocols or even used on other protocols. These abstractions are particularly useful for enumerative automated verification, where one needs to cope with the problem of large state spaces. We have built a significant support for automated analysis of RIP and AODV, which includes their machine readable specifications.

The work described in this thesis can be extended in several directions. In the area of protocol results, BGP is probably the most interesting target. Currently, there are many open research and engineering problems related to its convergence. In particular, scalability of BGP seems to be a growing practical concern, given the rapid growth of the Internet. Besides studying behavior of the *current* BGP, the researchers are interested in the ways to improve it. We have not discussed this topic, since it surpasses the scope of the thesis, which is about analysis (and not design) of routing protocols.

On the tool side, one direction is building custom tools for correctness analysis of routing protocols. The restricted domain would probably allow improvements of the techniques used in general-purpose tools. A particularly interesting area within this direction would be integration of simulation, testing and verification tools for the purpose of analyzing routing protocols.

Another direction is work on reusable components. It would be useful to build a uniform framework in which proofs and models could be reused across different protocols. For instance, the same network model (which, among other things, includes assumptions about topology and reliability of links) is often used for many routing protocols. One should be able to construct that model once and then reuse it in different instances. Similar components could be built for traffic models.

Finally, it would be worth to study how to efficiently integrate protocol design with formal analysis. Efficient communication between designers and verifiers benefits both sides—it makes the

verification efforts worthwhile and routing protocols better.

Appendix A

Code Samples

A.1 RIP Pseudo-code

process RIPRouter

state:

me	// ID of the router
interfaces	// Set of router's interfaces
known	// Set of destinations with known routes
$hops_{dest}$	// Distance estimate
$nextRouter_{dest}$	// Next router on the way to *dest*
$nextIface_{dest}$	// Interface over which the route advertisement was received
timer $expire_{dest}$	// Expiration timer for the route
timer $garbageCollect_{dest}$	// Garbage collection timer for the route
timer *advertize*	// Timer for periodic advertisements

initially:

```
{
  known ← the set of all networks to which the router is connected.
  for dest ∈ known
  {
    hops_dest = 1
    nextRouter_dest = me
    nextIface_dest = the interface which connects the router to dest.
  }
  set advertize to 30 seconds
}
```

events:

```
  receive RIP (router, dest, hopCnt) over iface
  timeout (expire_dest)
  timeout (garbageCollect_dest)
  timeout (advertize)
```

utility functions:

```
broadcast(msg, iface)
{
  Broadcast message msg to all the routers attached to the network on the other side
  of interface iface.
}
```

event handlers:

```
receive RIP (router, dest, hopCnt) over iface
{
  newMetric ← min (1 + hopCnt, 16)
  if (dest ∉ known) then
  {
    if (newMetric < 16)
    {
      hops_dest ← newMetric
      nextRouter_dest ← router
      nextIface_dest ← iface
      set expire_dest to 180 seconds
      known ← known ∪ {dest}
    }
  } else
  {
    if (router = nextRouter_dest) or (newMetric < hops_dest)
    {
      hops_dest ← newMetric
      nextRouter_dest ← router
      nextIface_dest ← iface
      set expire_dest to 180 seconds
      if (newMetric = 16) then
      {
        set garbageCollect_dest to 120 seconds
      } else
      {
        deactivate garbageCollect_dest
      }
    }
  }
}

timeout (expire_dest)
{
  hops_dest ← 16
  set garbageCollect_dest to 120 seconds
}

timeout (garbageCollect_dest)
{
  known ← known − {dest}
}
```

```
timeout (advertize)
{
  for each dest ∈ known do
    for each i ∈ interfaces do
    {
      if (i ≠ nextIface_dest) then
      {
        broadcast ([RIP(me, dest, hops_dest)], i)
      } else
      {
        broadcast ([RIP(me, dest, 16)], i)        // Split horizon with poisoned reverse
      }
    }
    set advertize to 30 seconds
}
```

A.2 AODV Pseudo-code

```
process AODVRouter

constants:
  RREP_WAIT_TIME                  // Set as described in the standard.
  ACTIVE_ROUTE_TIMEOUT            = 3000 milliseconds
  MY_ROUTE_TIMEOUT                = 6000 milliseconds
  BAD_LINK_LIFETIME               = 2 * RREP_WAIT_TIME
  REV_ROUTE_LIFE                  = RREP_WAIT_TIME
  BCAST_ID_SAVE                   = 30000 milliseconds

state:
  me                      // ID of the router
  mySeqno                 // Router's own sequence number
  myBcastID               // Router's current broadcast ID
  known                   // Set of destinations with known routes
  sn_d                    // Destination sequence number
  hops_d                  // Distance in hops
  next_d                  // Next hop
  neighbors               // Set of all neighbors
  active_d                // Set of active neighbors
  timer lifetime_d              // Route expiration timer
  timer activeTimer_dest,n      // Active neighbor timer


// Initialization was missing in the original standard
initially:
{
  mySeqno ← 0
  myBcastID ← 0
  known ← ∅
  ∀d. sn_d ← 0
```

}

events:

receive RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno) from sender
receive RREP(hopCnt, dest, destSeqno, lifetime); IP_DEST from sender
receive NChange // Triggered when the set of neighbors change
receive Packet; IP_DEST from sender // Triggered when a the router receives a packet
// to forward to IP_DEST
timeout ($lifetime_{dest}$) // Triggered when $lifetime_{dest}$ times out
timeout ($activeTimer_{dest,n}$) // Triggered when $activeTimer_{dest,n}$ times out

utility functions:

seen (*source, bcastID*)
{
Determines whether a RREQ from *source* with the same or more recent broadcast ID as
bcastID has already been received by the router within the last *BCAST_ID_SAVE* milliseconds.
}

updateRoute (*dest, destSeqno, hopCnt, nextHop, ltime*)
{
Update the routing table with a new route to *dest*, which is *hopCnt* hops long,
continues via *nextHop* and has the attached destination sequence number *destSeqno*.
If no previous route to *dest* exists or if the new route is better than a previously existing one,
install the new route with $lifetime_{dest}$ timer set to *ltime* and include *dest* in *known*.
}

updateTable ()
{
Invalidate all entries in the routing table that use a non-neighbor as their *nextHop*
by setting their *hops* to infinity and their expiration timers to *BAD_LINK_LIFETIME*.
}

broadcast (*msg*)
{ Broadcast the message *msg* to all neighboring nodes. }

neighborcast (*msg, n*)
{ Send the message *msg* to the neighbor *n*. }

computeNeighbors ()
{ Return the current (most recent) set of neighbors. }

event handlers:

receive RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno) from sender
{
if not *seen*(*source, bcastID*)
{
$hopCnt \leftarrow hopCnt + 1$
if ($dest = me$) **then**
{
updateRoute (*source, sourceSeqno, hopCnt, sender, ACTIVE_ROUTE_TIMEOUT*)

```
      neighborcast ([RREP(0, me, max(mySeqno, destSeqno), MY_ROUTE_TIMEOUT); source], next_source)
    } else
    {
      updateRoute (source, sourceSeqno, hopCnt, sender, max(REV_ROUTE_LIFE, lifetime_dest))
      if (dest ∈ known) and (hops_dest < ∞) and (sn_dest ≥ destSeqno) then
      {
        neighborcast ([RREP(hops_dest, dest, sn_dest, lifetime_dest); source], next_source)
        n ← next_dest
        active_source ← active_source ∪ {n}
        set activeTimer_source,n to ACTIVE_ROUTE_TIMEOUT
      } else
      {
        broadcast ([RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno)])
      }
    }
  }
}

// This handler was missing in the original standard
// We believe that the following code describes the desired functionality:
 receive RREP (hopCnt, dest, destSeqno, lifetime); IP_DEST from sender
 {
   if (IP_DEST = me) then
   {
     if (hopCnt = ∞) then
     {
       updateRoute (dest, destSeqno, ∞, sender, BAD_LINK_LIFETIME)
       for n ∈ active_dest
       {
         neighborcast ([RREP (∞, dest, destSeqno, BAD_LINK_LIFETIME); n], n)
       }
     }
     else
     {
       updateRoute (dest, destSeqno, hopCnt, sender, lifetime)
     }
   }
   else
   {
     updateRoute (dest, destSeqno, hopCnt, sender, lifetime)
     neighborcast ([RREP(hopCnt + 1, dest, destSeqno, lifetime); IP_DEST], next_IP_DEST)
   }
 }

 receive NChange
 {
   newNeighbors ← computeNeighbors ()
   disconnected ← neighbors − newNeighbors
   neighbors ← newNeighbors
   mySeqno ← mySeqno + 1
   for dest ∈ known
```

```
  {
    if (next_dest ∈ disconnected)
    {
      for n ∈ active_dest
      {
        neighborcast ([RREP(∞, dest, 1 + sn_dest, BAD_LINK_LIFETIME); n], n)
        sn_dest ← sn_dest + 1   // This line was missing in the original standard
      }
    }
  }
  updateTable ()
}

receive Packet; IP_DEST from sender
{
  if (IP_DEST ≠ me)
  {
    if (IP_DEST ∈ known) then
    {
      active_IP_DEST ← active_IP_DEST ∪ {sender}
      set lifetime_IP_DEST to ACTIVE_ROUTE_TIMEOUT
      set activeTimer_IP_DEST,sender to ACTIVE_ROUTE_TIMEOUT
      neighborcast ([Packet; IP_DEST], next_IP_DEST)   // Forward the packet towards IP_DEST
    } else
    {
      myBcastID ← myBcastID + 1
      broadcast ([RREQ (0, myBcastID, IP_DEST, sn_IP_DEST, me, mySeqno)])
      Queue the packet and forward it upon establishing a route to IP_DEST.
    }
  }
}

timeout (lifetime_dest)
{
  if (hops_dest = ∞) then
  {
    Mark the entry for dest as "erasable". Erasable entries can be garbage collected.
    Garbage collecting sets sn_dest to 0, hops_dest and next_dest to some undefined value.
  } else
  {
    hops_dest ← ∞
    known ← known − {dest}
    set lifetime_dest to BAD_LINK_LIFETIME
  }
}

timeout (activeTimer_dest,n)
{
  active_dest ← active_dest − {n}
}
```

A.3 AODV Modification

The new handler for route expiry is simpler—it only disables the route and increases the sequence number.

timeout ($lifetime_{dest}$)
{
  $hops_{dest} \leftarrow \infty$
  $sn_{dest} \leftarrow sn_{dest} + 1$
  $known \leftarrow known - \{dest\}$
}

A.4 SPVP Pseudo-code

process SPVP
state:

me	// ID of the router
$peers$	// The set of neighbors (BGP peers)
rib_{dest}	// Current route to $dest$
$ribIn_{p,dest}$	// Last advertized route to $dest$ from the peer p

events:
  receive UPDATE $(dest, path)$ *from* p

utility functions:
  $best(dest)$
  {
    $choices \leftarrow \{\epsilon\} \cup \{(me\ p)ribIn_{p,dest} \mid p \in peers\}$
    if (dest=me)
    {
      $choices \leftarrow choices \cup \{(me)\}$
    }
    Return the most preferred route from the set $choices$.
  }

  $send(msg, p)$
  { Send the message msg to the peer p. }

initially:
{
  for each $p \in peers$ **do**
  {
    $\forall dest.\ ribIn_{p,dest} \leftarrow \epsilon$
  }
  $rib_{me} \leftarrow (me)$
  for each $p \in peers$ **do**
  {
    send $([UPDATE(me, rib_{me})], p)$
  }

```
}

event handlers:
  receive UPDATE (dest, path) from p
  {
    ribIn_{p,dest} ← path
    b ← best(dest)
    if (rib_dest ≠ b)
    {
      rib_dest ← b
      for each v ∈ peers do
      {
        send ([UPDATE(dest, rib_dest)], v)
      }
    }
  }
```

Bibliography

[1] Mart´ın Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:73–132, 1993.

[2] Home page for the ACL2 theorem proving system. `http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html`.

[3] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Ramajani, and S. Tasiran. MOCHA: modularity in model checking. In *10th International Conference on Computer-Aided Verification*, pages 516–520. LNCS, Springer-Verlag, 1998.

[4] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 1644. LNCS, Springer-Verlag, 1999.

[5] Dimitri P. Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1991.

[6] Karthikeyan Bhargavan, Carl Gunter, and Davor Obradovic. Formal verification of standards for distance vector routing protocols. Submitted to the Journal of ACM, 2000.

[7] Karthikeyan Bhargavan, Carl A. Gunter, Elsa L. Gunter, Michael Jackson, Davor Obradovic, and Pamela Zave. The village telephone system: A case study in formal software engineering. In Jim Grundy and Malcolm Newey, editors, *Proceedings of TPHOLs '98 Conference*, Canberra, Australia, September 1998.

[8] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations. In *International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, 2000.

[9] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. Fault origin adjudication. In *Formal Methods in Software Practice (FMSP)*, Portland, OR, 2000.

[10] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. RIP in SPIN/HOL, August 2000. Theorem Provers for Higher-Order Logics.

[11] Home page for the Coq proof assistant. `http://coq.inria.fr/`.

[12] V. Dimitrov and A. Petkov. Using petri nets for communication protocol verification. *Wissenschaftliche Beitr¨age zur Informatik, Informatik-Zentrum an der TU Dresden, Heft 3*, pages 44–46, 1987.

[13] Amy P. Felty, Douglas J. Howe, and Frank A. Stomp. Protocol verification in nuprl. In *Proceedings of International Conference on Computer-Aided Verification (CAV)*, June 1998.

[14] Amy P. Felty and Frank A. Stomp. A correctness proof of a cache coherence protocol. In *Proceedings of 11th Annual Conference on Computer Assurance*, June 1996.

[15] Drew Fudenberg and Jean Tirole. *Game Theory*. MIT Press, 1991.

[16] Lixin Gao. On inferring autonomous system relationships in the internet. *IEEE Global Internet*, November 2000.

[17] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. In *ACM SIGMETRICS*, 2000.

[18] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[19] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. Policy disputes in path-vector protocols. In *Proceedings of ICNP '99 Conference*, Toronto, Canada, October 1999.

[20] Timothy G. Griffin and Gordon Wilfong. An analysis of BGP convergence properties. In Guru Parulkar and Jonathan S. Turner, editors, *Proceedings of ACM SIGCOMM '99 Conference*, pages 277–288, Boston, August 1999.

[21] Timothy G. Griffin and Gordon Wilfong. A safe path vector protocol. In *Proceedings of INFOCOM 2000 Conference*, Tel Aviv, Israel, March 2000.

[22] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, May/June 2000.

[23] C. Hendrick. Routing information protocol. RFC 1058, IETF, June 1988.

[24] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In *Computer Aided Verification (CAV)*, 1998.

[25] Home page for the HOL interactive theorem proving system. `http://www.cl.cam.ac.uk/Research/HVG/HOL`.

[26] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[27] Gerard J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[28] John W. Stewart III. *BGP4 (Inter-Domain Routing in the Internet)*. Addison-Wesley, 1998.

[29] Audun Joesang. Security protocol verification using spin. In *SPIN Workshop*, Montreal, Quebec, October 1995.

[30] K.L.McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[31] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. An experimental study of internet routing convergence. Technical Report MSR-TR-2000-08, Microsoft Research, 2000.

[32] C. Labovitz, G. R. Malan, and F. Jahanian. Internet routing instability. In *Proceedings of ACM SIGCOMM '97 Conference*, Cannes, France, September 1997.

[33] C. Labovitz, G. R. Malan, and F. Jahanian. Origins of internet routing instability. In *Proceedings of IEEE INFOCOM '99 Conference*, New York, USA, March 1999.

[34] Craig Labovitz, Roger Wattenhofer, Srinivasan Venkatachary, and Abha Ahuja. The impact of internet policy and topology on delayed routing convergence. In *Proceedings of INFOCOM 2001*, Anchorage, Alaska, April 2001.

[35] G. Malkin. *RIP Version 2 Carrying Additional Information*. IETF RFC 1388, January 1993.

[36] G. Malkin. Rip version 2 applicability statement. RFC 1722, IETF, November 1994.

[37] G. Malkin. Rip version 2 carrying additional information. RFC 1723, IETF, November 1994.

[38] G. Malkin. Rip version 2 MIB extension. RFC 1724, IETF, November 1994.

[39] G. Malkin. Rip version 2 protocol analysis. RFC 1721, IETF, November 1994.

[40] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[41] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Seventh USENIX Security Symposium*, pages 201–216. USENIX, San Antonio, 1998.

[42] J. Moy. OSPF version 2. RFC 1583, IETF, March 1994.

[43] Vern Paxson. End-to-end routing behavior in the internet. In *Proceedings of ACM SIGCOMM '96 Conference*, Stanford University, USA, August 1996.

[44] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on demand distance vector (AODV) routing. Internet-Draft Version 2, IETF, March 1998.

[45] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, pages 90–100, February 1999.

[46] Home page for the PVS specification and verification system. `http://pvs.csl.sri.com/`.

[47] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *IFIP Working Conference on Programming Concepts and Methods (PROCOMET '98)*, Shelter Island, NY, June 1998.

[48] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). RFC 1771, IETF, March 1995.

[49] Mark A. Smith. Formal verification of Communication Protocols. In *Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE/PSTV) IFIP*, pages 129–144, October 1996.

[50] Home page for the SMV model checker. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[51] Home page for the SPIN model checker. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[52] K. Varadhan, R. Govindan, and D. Estrin. Persistent route oscillations in inter-domain routing. ISI Technical Report 96-631, USC/Information Sciences Institute, 1996.

[53] C. Villamizar, R. Chandra, and R. Govindan. BGP route flap damping. RFC 2439, IETF, November 1988.