Formal Methods for Modeling and Verifying an Ad hoc network protocol

Marina Gelastou

June 5, 2007

Acknowledgements

With this work I complete my graduate studies. Therefore I would like to express my gratitude to my thesis supervisors, Assistant Professor Anna Philippou and Lecturer Chryssis Georgiou for their support during my studies and for the things I've learned from both of them.

Special thanks to Mr. Theophanis Pavlides for our cooperation in the early steps of this work and to Dr. Vicky Papadopoulou for her precious advice.

Finally, lots of thanks to my family and my friends for being next to me all these years encouraging me and supporting me. Their patience and love helped me overcome the difficulties of the graduate programme.

Abstract

This dissertation is an investigation into the application of two formal methods in the area of ad hoc networks. In ad hoc networks, nodes are free to move, changing in this way their topology dynamically. This dynamic nature increases the complexity of the algorithms designed for ad hoc network and the verification of such algorithms is a difficult error-prone task that requires much effort and ingenuity. Thus, it is a field where formal verification has a lot to offer.

Even though the field of ad hoc networks received a lot of attention by the research community, no formal method specialized for this kind of environments was proposed until recently. This dissertation aims to study the characteristics of ad hoc networks and employ two popular formal methods, I/O Automata and Process Algebra, to model and analyze an algorithm of this area in order to investigate the foundations of the modeling, development and analysis of ad hoc networks.

In this work, the two frameworks are reviewed, presenting their definition and their verification methods. Moreover, a study of ad hoc networks specifies their important features along with the problems arising in this area. An algorithm of this field is identified to be used for specification and verification purposes. This algorithm is named "Ad hoc On-demand Multipath Distance Vector Routing" and it has not been verified previously. The algorithm is specified using the languages of the process algebra CCS and the basic I/O Automata framework and its correctness is proved using standard techniques originating from the formalisms.

Finally, the two formalisms are compared with respect to their applicability in the area of ad hoc networks, according to the observations made during the algorithm's modeling and analysis.

Contents

Chapter 1

Introduction

1.1 Motivation

The widespread use of computer systems and networks in the last decades has been accompanied by various problems regarding reliability, security and efficiency. Problems caused by innocent errors in software, sometimes can have costly or even catastrophic outcome. Characteristic problems that have arisen in the field of networks include the security flaws detected in the 802.11 and Bluetooth protocols. The demands for reliability from today's computing systems, especially the safety-critical ones, renders the need of scientific methodologies for the development of reliable algorithmic solutions for these computational environments widely acknowledged. In fact, it appears as one of the "Grand Challenges" in the proceedings of the "Grand Challenges in Computing Research" conference (Newcastle, March 2004). This dissertation aims to investigate the foundations of the modeling, development and analysis of ad hoc networks.

Ad-hoc networks [47] are a technology with many applications in the sectors of health, business and communications in general. An ad-hoc network consists of a set of hosts that constitute a self-configuring network in the absence of a fixed infrastructure or central administration. Each host of an ad-hoc network acts as a router forwarding packets to other nodes [20]. Such a network may operate in a standalone fashion, or may be connected to the Internet. Minimal configuration

and quick deployment make ad hoc networks suitable for emergency situations like natural or human-induced disasters, emergency medical situations, etc.

An important challenge in such networks is the development of algorithmic solutions to problems arising therein which respond to the dynamicity of network nodes while conserving the network resources. Until recently, the correctness of these protocols was checked, in general, using informal algorithms and via testing. In this way wide margins of errors were allowed, risking the existence of safety and reliability problems.

In the last two decades, the field of formal methods for system design and analysis has dramatically matured and has reported significant success in the development of theoretical frameworks for formally describing and analyzing complex systems. More specifically, significant research efforts were geared towards the development of formal methodologies for system modeling and verification. One such model is that of Process Algebra, PA [8, 18]. PAs constitute a formal framework with well-defined semantics which allows the compositional modeling and analysis of concurrent systems. They are equipped with precise semantics, providing a solid basis for understanding system behavior and for reasoning about their correctness. The model of Input/Output Automata of Lynch and Tuttle [33] is another popular formal framework for modeling distributed systems. I/O Automata contain a number of mathematic tools and methodologies which allow carrying out correctness proofs and precise analysis of complex static and dynamic distributed algorithms.

Recently some formal methods have been proposed for the field of ad hoc networks. The $CBS^{\sharp}$ [43] calculus enriches the Calculus of Broadcasting Systems (CBS) [16] with the notion of locations as they are modeled in Mobile Ambients [10] and cryptographic primitives in order to model and analyze security issues. The CMN [37] calculus has a two-level syntax, one for processes and one for networks. The processes are accompanied with their physical location and their transmission range which define their cell. There is no broadcasting channel and the processes can communicate through channels in their cell. CMN also models node movements. The CMAN [21] calculus also has a two level syntax, it contains the notion of location (logical) and node movements but has no channels. A node communicates with other nodes only if they are in specific locations. In the area of I/O Automata, the Dynamic I/O Automata [3] were proposed for dynamic systems in general and can be applied in ad hoc networks. Each automaton can change behavior depending to their environment, modeling in this way the dynamic behavior of a system.

These methods aim to model features of ad hoc networks like broadcast com-

munication and dynamic nature (Section 3.1) that simple methods like CCS [38] and general I/O Automata [32] do not model directly. However, in this work we choose to use simple methods for three main reasons. The first reason is that both CCS and I/O Automata are mature methods with well established analysis methods and techniques. The second reason is that we want to study how ad hoc networks affect these formalisms. Studying them in the primitive forms of the two formalisms we will get a deep understanding of the effects and needs that have to be fulfilled by a new method for this area. And finally, the complexity of these models is higher than the simple methods. Ad hoc networks themselves are systems with high complexity, thus adding complexity from the model will make the analysis of these systems more difficult. Using simple methods we keep the complexity of these systems as low as possible.

This dissertation focuses on the specification and verification of a multipath routing algorithm in ad hoc networks, named "Ad hoc On-demand Multipath Distance Vector Routing" [36] via both I/O Automata and Process Algebra formalisms in order to illustrate how the dynamic nature of ad hoc networks affects the application of the existing formal methods in this field. Finally a comparison of the two methodologies with respect to their applicability in the field of ad hoc networks aims to identify the strengths and weaknesses of the two frameworks in the field under study.

1.2 Related work

The field of networks and especially the routing problem has received a lot of interest because of their significance. Moreover, the need of providing higher assurance in the execution of the proposed algorithms has prompted researchers to analyze these algorithms with the aid of formal methods. [26] and [9] are two examples of formal verification of routing protocols BGP and RIP respectively, routing protocols of traditional networks. Moreover, in [9] and [11] formal methods are used for analyzing the routing protocol AODV from the field of ad hoc networks while in [42] the authors analyze another algorithm of ad hoc networks, the DSR protocol. These works concern applications of variations of process algebras for each field.

I/O Automata have also been employed for the specification and verification of ad-hoc network algorithms, like in [14, 13] and [15].

1.3 Thesis Structure

The rest of this dissertation is organized as follows. The second chapter, presents a summary of the I/O-Automata and the process-algebraic frameworks which will be used for the specification and verification of the ad hoc networks protocol. Chapter 3 presents the characteristics and problems arising in ad hoc networks, as well as the algorithm AOMDV which was selected for the specification and verification purposes of this work. The next two chapters discuss the specification and verification of AOMDV using the I/O Automata (Chapter 4) and the process-algebraic (Chapter 5) frameworks. Chapter 6, which concludes this dissertation, discuss the strengths and weaknesses of the two frameworks according to their applicability in the field of ad hoc networks.

Chapter 2

Formal Methods

In this chapter we present the frameworks for specification and verification that are used in this work. We first present I/O Automata, followed by the presentation of Process Algebras.

2.1 I/O Automata

An *Input/Output Automaton* [32], or I/O Automaton, is a powerful tool for modeling asynchronous distributed systems and algorithms. This modeling can be used as a formal basis tool for algorithm correctness proof, carrying out complexity analysis, proving upper and lower bounds on the complexity of solving particular problems, and proving impossibility results.

2.1.1 Definition of I/O Automaton

An I/O Automaton is a state machine which can be infinite and nondeterministic. It has three types of transitions, its *actions*, which are classified as *input*, *output* and *internal*. The input actions of an I/O Automaton are generated by the environment and are transmitted instantaneously to the automaton. In contrast, the

automaton can generate the output and internal actions autonomously and can transmit output actions instantaneously to its environment. A *signature* S of an I/O Automaton A, denoted by $sig(A)$, is a triple of the disjoint sets: Input actions $in(S)$, Output actions $out(S)$ and Internal actions $int(S)$ of the automaton. The set of all actions is denoted by $act(S)$. The set of the external actions of an I/O Automaton given a signature S is denoted as $ext(S)$ and it is the union of the input and output actions of that signature S. In other words it is the set of the actions that are visible to the environment. An external action signature is an action signature S with no internal actions, and is denoted by $extsig(S)$.

The actions of an I/O Automaton are given in a precondition-effect style, where the preconditions give the conditions that must be satisfied in order to enable the action. Since *input* actions are generated by the environment, they are not controlled by the automaton, thus, they don't have preconditions and they are always enabled. This feature is the most important characteristic of I/O Automata. The set of actions of an I/O Automaton that are controlled by that automaton denoted by $local(S)$, is the union of the internal and output actions of some given signature S of that automaton.

Formally an I/O Automaton A consists of the following:

- an action signature $sig(A)$,
- a set $states(A)$ of states,
- a nonempty set $start(A) \subset states(A)$ of start states,
- a transition relation $trans(A) \subset states(A) \times acts(A) \times states(A)$ with the property that for every state s and input action π there is a transition $(s, \pi, s') \in trans(A)$ to some state s', also called a *step*, and
- an equivalence relation $tasks(A)$ partitioning the set $local(A)$ into different sets each one containing the local actions that are in control of one system component.

Executions and traces of an I/O Automaton. An *execution fragment* of an automaton A is a finite sequence $s_0, \pi_1, s_1, \cdots \pi_n, s_n$ or an infinite sequence $s_0, \pi_1, \cdots$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1}) \in trans(A)$ for all $i \geq 0$. An *execution* is an execution fragment that starts with a start state (i.e. $s_0 \in start(A)$). We denote as $trace(\alpha)$ of some execution α the sequence that is obtained by restricting α to the set of external actions of A. We denote the set of all traces of A by $traces(A)$.

Composition of I/O Automata. I/O Automata can be composed to create more complex I/O Automata. During this composition the same-named actions of the different automata are identified and if an automaton A has an output action π and one or more automata have as input that same action π, then these automata are composed together. In this way when A generates the output action π, it instantaneously transmit this action to all automata that have the input action π, while the rest of the automata in the composition do nothing.

The above suggests that in order for two automata A and B to be composed, the internal actions of A must form a disjoint set with the actions of B since automaton's A internal actions can force actions from B to execute. Additionally, since at most a single system component controls the performance of any given action, there can not be a composition of two automata A and B unless the output actions of A and B form disjoint sets. Finally since the composition is allowed to take place in any countable collection of I/O Automata, an action of a composition must be an action of only finitely many of the compositions components. This restriction exists so the possibility that an action π that exists in infinitely many automata, will require an infinite amount of work in order to be performed.

Fairness. Since the input of an I/O Automaton is not controlled by the automaton itself but by its environment, an I/O Automaton can exhibit arbitrary behavior due to "bad" input from the environment. The I/O Automaton specification does not block the automaton to exhibit such arbitrary behavior but considers only the rational executions of the automaton.

Such "bad" input of an I/O Automaton could result in executions that are not *fair*. Fair is the property of an automaton that infinitely often it has the opportunity to perform one of its local actions. Then a *fair execution* is an infinite execution that all the system components are fair.

Extensions of I/O Automata. Over the I/O Automata there have been developed some variations of I/O Automata that concern different aspects of concurrent systems, like time, probabilistic behavior and dynamic nature. These are:

1. The *Timed I/O Automata* [27] which are used mainly for describing and analyzing real-time concurrent systems and time-dependent systems in general.

2. The *Hybrid I/O Automata* [31] that find applications in embedded systems.

3. The *Probabilistic I/O Automata* [30] that are appropriate for modeling and analyzing probabilistic systems.

4. The *Dynamic I/O Automata* [3] where each automaton is able to change its behavior (i.e. its signature S), making them appropriate for use in dynamic systems where the components change their behavior depending their environment.

For this work's purpose, we prefer the general model of I/O Automata because none of these extensions can give more than the general model since ad hoc networks are not embedded systems, neither their components change their behavior. Moreover, the algorithm that is described and verified in this work does not involve time or probabilistic issues.

2.1.2 Verification Methods used in I/O Automata

I/O Automata can be used not only for precise description of asynchronous systems but also to formulate and prove precise claims about what systems do.

Safety-Liveness Properties In I/O Automata the correctness proof of an automaton is usually deduced to showing *safety* and *liveness properties* of the automaton. Informally speaking, the former requires that some particular "bad" thing never happen, which in return means that if something bad happens in a trace, then it happens as a result of some particular event in the trace. The latter requires that something "good" will eventually happen, which in return means that no matter what has happened up to this point, there is still the possibility that something good will happen. Liveness properties can only be satisfied by fair executions. Taking these properties together one can prove claims such as "Eventually the system will exhibit some required behavior".

Invariants Another important property to show is an *invariant assertion*, or just an *invariant*, of some automaton A. Invariant is a fundamental property that is true in all reachable states of A. Invariants are typically proved by induction on the number of steps in an execution leading to the state in question. More generally, invariants are used to prove other invariants which in turn are used when carrying out subsequent inductive proofs. By proving that an automaton A is described by specific invariant I, it is proved that automaton A exhibits the same behavior as that invariant. Several invariants are usually combined in proving safety properties of a given automaton.

When dealing with composition of many automata to a more complex automaton, it is often easier to reason about the automata individually. First, less complex

invariants for those automata are proved, and then the composition of those invariants is used to reason about the composed automaton.

Hierarchical Proofs. Automata are used to describe complex systems but proving the correctness of such systems is usually difficult. A good proof strategy is based on hierarchy of automata. This hierarchy represents a series of descriptions of a system or algorithm at different levels of abstraction. The proof begins at the highest level of abstraction where there is the specification of an automaton, and through successive refinement the proof continues to lower levels of abstraction by introducing more of the detail that exists in the final system or algorithm. Because of the extra detail, lower levels of abstraction are usually harder to understand than higher levels. In order, now, to prove properties of the lower level, these lower level automata are related to the higher level automata.

In order to establish the correspondence between two different automata at different levels of abstraction some *simulation techniques* are used. A simulation proof between two automata A and B that have the same external signature, known as a *simulation*, involves establishing a correspondence between A and B. The existence of a simulation between A and B is used to show that any behavior that can be exhibit by A, can also be exhibit by B. This means that if B solves a particular problem, so does A.

Now, if A is regarded as an automaton at a lower level of abstraction, and B as an automaton at a higher level of abstraction, with the help of a simulation it can be established that a more detail version of the problem, exhibits the same behavior as the automaton at a higher level of abstraction. Since the behavior of an automaton is its trace, thus A simulates B iff $trace(A) \subseteq trace(B)$. Repeating this process to even more abstract levels where it is easier to prove the correctness of an automaton, it is established that the implementation of the problem does is fact solves the specific problem.

There are six kinds of simulations: The simplest type of these simulations is the *refinement*. A refinement is a simulation between two automata A and B where the image of every start stage of A is a start stage of B, and every step of A has a corresponding sequence of steps of B that begins and ends with the images of the respective beginning and ending states of the given step, and has the same external actions.

The second and third types of simulations are the *forward simulations* and *backward simulations* respectively. A forward simulation between A and B is a generalization of a refinement where in the image of every start state of A there is

a start state of B, and a forward step in A can be simulated from corresponding states in B. This generalization of the refinements allows a set of B to correspond to a single state of A.

A backward simulation which is another generalization of refinements is the dual of a forward simulation. A backward simulation is a simulation between A and B where every image of every start state of A must be a start state of B, and every step of B is simulated from corresponding states of A.

The next type of simulations are the *hybrid simulations* which is a combination of a forward and a backward simulation, where the order of the simulation can be either a forward-backward, or a backward-forward.

The last two types of simulations are the *history* and *prophecy simulations*. A history simulation is in fact a forward simulation between A and B which its inverse is a refinement between B and A. Similarly a prophecy simulation is a backward simulation from A to B which its inverse is a refinement from B to A.

I/O Automata toolkit. Work has been carried out in the production of automated verification of systems that can validate a given system as to where it exhibits a certain behavior. Such tools take as input the specification of an automaton and they aid in the system development, testing, and verification.

Specifically the I/O Automata toolkit [1] mainly consists of two parts. The first part is the front-end tool (*semantic checker*) that checks the syntax of the given automaton if it complies with the I/O Automata language syntax and semantics. The second part is the back-end tools that consists of a *simulator* for testing the behavior of I/O Automata, the *theorem provers* for automated theorem proving of invariants and other properties of automata, the *model checkers* and an *automated code generator* which produces java code for actual implementations.

2.2 Process Algebra

Process algebras are a well-established class of modeling and analysis formalisms for concurrent and distributed systems. They are high-level description languages that contain operators for building processes including constructs for defining recursive behaviors. They are accompanied by semantic theories which give precise meaning to processes, translating each process into a mathematical object on which rigorous analysis can be performed. In addition, they are associated with axiom systems which prescribe the relations between the various constructs and can be used to reason algebraically about processes. During the last two decades, they have been

extensively studied and they have been proved quite successful in modeling and reasoning about system correctness. They have been extended to model a variety of aspects of process behavior including value passing, distribution, mobility and asynchronous communication.

2.2.1 The Calculus of Communicating Systems

The *Calculus of Communicating Systems* [38], or CCS for short, is one of the first and most famous process algebras. It was developed by Robin Milner to model and analyze process interaction in concurrent systems. In CCS computational entities are defined by the notion of a process and communication between processes is achieved by sending and receiving signals through common channels.

Specifically, CCS assumes a set of channels Λ, ranged over by a and b. Channels provide the basic communication and synchronization mechanisms in the language. A channel a can be used in input position, denoted by a or in output position, denoted by $\bar{a}$. This gives rise to the set of actions Act of the calculus, ranged over by α and β, containing the set of *input actions*, the set of *output actions* and the *internal action*. An input action over a channel a is denoted as a, an output action over the same channel is denoted as $\bar{a}$ while the internal action is denoted as τ.

Let $\alpha \in Act$ be an action, $L \subset \Lambda$, $f : L \rightarrow \Lambda$ a relabeling function over channels and C a set of processes. The syntax of CCS processes is given by the following grammar:

$P \ ::= \ 0 \mid a.P \mid P + Q \mid P|Q \mid P\backslash L \mid P[f] \mid C$

Process 0 is the inactive process, $\alpha.P$ represents the process that first executes action α and then proceeds as P. $P + Q$ is the choice operator, and $P|Q$ is the concurrent composition of P and Q. $P\backslash L$ represents the process P with the scope of channels in L restricted within P. $P[f]$ is the relabeling of the channels in P according to the function f and C is a process constant. Each such constant is associated with a definition of the form $C \stackrel{\text{def}}{=} P$, where the process P may contain occurrences of C as well as other process constants.

Each operator of CCS is associated with a set of rules which are presented in Figure 2.1.

CCS is a very expressive process algebra. It models directly concurrent and distributed systems. Moreover, CCS can be easily extended in order to express many features of systems like value passing and conditional statements (if-then-else).

Operational semantics of CCS

Action rule (Act): $\frac{-}{\alpha.P \xrightarrow{\alpha} P}$

Choice rule 1 (Sum_1): $\frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$

Choice rule 2 (Sum_2): $\frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$

Composition rule 1 ($Comp_1$): $\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$

Composition rule 2 ($Comp_2$): $\frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'}$

Composition rule 3 ($Comp_3$): $\frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\bar{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$

Restriction rule (Res): $\frac{P \xrightarrow{\alpha} P'}{P \backslash L \xrightarrow{\alpha} P' \backslash L}$ $\alpha, \bar{\alpha} \notin L$

Relabeling rule (Rel): $\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{\hat{f}(\alpha)} P'[f]}$

Figure 2.1: Operational semantics of CCS

Process Algebra variations Due to its simplicity, expressiveness and extensibility, CCS became the basis framework for many extensions that were proposed to satisfy different needs that cannot be modeled directly by CCS. Here, there is a list of some Process Algebras that extend CCS to model mobility, dynamic behavior and broadcasting.

1. The most famous extension of CCS is the π-calculus [40] which models mobility introducing the notion of names and name-passing. A name can be a variable or a channel and processes communicate by exchanging those names in order to model systems with dynamically-evolving topology.

2. CBS [16] extend CCS to model broadcasting. It has no channels and communication between different nodes is achieved through a common, global broadcasting channel.

3. The Ambient calculus [10] is another CCS extension that adds boundaries in its semantics. Ambients are processes that have a restricted environment to act where they can enter, exit, and open other ambients in order to move or secure information. Communication between ambients is not based on

channels as in CCS and can exists only between ambients in the same ambient. Ambient calculus is appropriate for modeling systems with agent mobility and security issues.

4. A restricted environment is also met in the $D\pi$- calculus [25] under the name "locations" where resources and agents coexist. Agents can move from one location to the other providing mobility, but the resources of the locations are static. Communication between two nodes can be achieved only if agents are at the same location.

5. A similar to the $D\pi$-calculus process algebra d-calculus [55] where locations are able to move. The agents in a location move with the location, not independently as in the D-calculus. An addition of this process algebra is the definition of capabilities and restrictions on channels. Moreover, new channels and locations can be declared and old channels and locations can be destroyed in the system to giving dynamic behavior.

6. In the Nomadic π-calculus [56] agents are located at particular sites and communication between two agents is achieved by sending a message to the receiver without specifying a channel. A new feature found in Nomadic the π-calculus is the location independent communication in addition to location dependent communication (agents has to be at the same site).

7. A special class of process algebras are the timed process algebras that can model time issues like timeouts and time passing. Timed process algebras include ATP [45] and TPL [24].

Most of the variations listed above provide useful operators for modeling mobility issues like movements, channel declaration and destruction which can be very helpful in ad hoc networks modeling. These process algebras were developed to model agent mobility, therefore they are not a natural choice for the field of ad hoc networks since they cannot directly model mobility of ad hoc networks. Moreover, timed process algebras are not appropriate since it appears that the timing needs posed by ad hoc network applications are simple (timeouts) and they can be simulated indirectly in untimed process algebras, for example, by defining special processes imitating a clock and issuing timeout messages when necessary. Therefore, CCS is more appropriate for this work, since it can be applied to model ad hoc networks features (even indirectly) and it can be easily extended with new operators if this is necessary.

More specifically, for this work, it was used an extension of CCS with value-passing and conditional decision (if-then-else). The conditional decision operator used in this work is the operator $\mathsf{cnd}\ (e_1 \ \triangleright \ P_1, \cdots, e_n \ \triangleright \ P_n)$ where all e_i are conditional terms. The process that contains the conditional operator behaves as P_i for the smallest i for which e_i is true.

2.2.2 Analysis Techniques

Process algebras provide a framework for modeling the behavior of concurrent and distributed systems. In addition, they are accompanied by a suite of techniques that allow us to prove that the model of a system is correct in that it fulfills its specification.

These techniques essentially perform analysis on the state-space of processes. These state spaces are given rise to by operational semantics defined for each of the above-mentioned process algebras. In particular, the operational semantics give precise meaning to each operator of a process algebra via a set of rules which, given a process P, prescribe the possible transitions the process may take. When a process takes a transition, it evolves into another process. The set of processes that a given process P can evolve into by performing a sequence of transitions defines the state space of P. The state space is represented as a labeled transition system with actions serving as labels.

The techniques that are described in this section are common proof techniques used for all process algebras.

1. Equivalence Checking Equivalence checking, in general, means to check that two processes exhibit equivalent behavior. In process algebras, it is used to establish that a system is correct by describing the system and its specification as two process-calculus processes and discovering an equivalence relation among them. Such equivalence relations are formally described through the notions of simulation and bisimulation [38]. Simulation and bisimulation are binary relations on states of systems.

Definition 2.2.1 *A binary relation $\mathcal{R}$ is a simulation if for each $(P, Q) \in \mathcal{R}$ and $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in \mathcal{R}$.*

Two processes are bisimilar if, for each step of one, there is at least one matching step of the other, leading to bisimilar states.

There exist many types of bisimulation, the two most popular ones being strong and weak bisimulation. Strong bisimulation demands that for each action presented

by the one process, there is at least one matching action presented by the other, leading to strongly bisimilar states. This provides that the two processes have exactly the same behavior. But sometimes, this is too restrictive since in practice one is interested in the observable behavior of processes and does not demand the matching of internal actions between them. Observational equivalence is based on the idea that two equivalent systems exhibit the same behavior at their interfaces with the environment. This idea is captured by weak bisimulation.

Definition 2.2.2 *Weak bisimulation is the largest symmetric relation, denoted by* $\approx$*, such that, if* $P \approx Q$ *and* $P \xrightarrow{\alpha} P'$*, then there exists* Q' *such that* $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ *and* $P' \approx Q'$*.*

where $\Longrightarrow$ is the reflexive and transitive closure of τ, $\stackrel{\alpha}{\Longrightarrow}$ is the composition $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$ and $\hat{\alpha} = \varepsilon$ if $\alpha = \tau$ or $\hat{\alpha} = \alpha$ otherwise.

Weak bisimulation abstracts away from internal computation by focusing on the external, observable actions. Two processes are weakly bisimilar if they can match each other's observable behavior. Note that even thought two weakly bisimilar processes have the same traces, that is they have the same observable behavior, the opposite does not necessarily hold. In other words, two processes with the same traces are not necessarily weakly bisimilar.

The theory of bisimulation relations has been developed into two directions. On one hand, axiom systems have been developed for establishing algebraically the equivalence between processes. On the other hand, proof techniques that ease the task of showing two processes to be equivalent have been proposed. Below is discussed the notion of confluence which has been associated with proofs techniques belonging to this latter type.

Confluence The notion of confluence was first studied by Milner [38]. Its essence, to quote [39] is that "of any two possible actions, the occurrence of one will never preclude the other". As shown in the mentioned papers, for pure CCS processes, confluence is preserved by several system-building operators. This fact makes it possible to reason compositionally that a system is confluent and to exploit this fact while reasoning about its behavior. In particular, it is often the case that for establishing the equivalence of a system with its specification, it is sufficient to establish conformance of only a single execution of the process: by the confluence of the process one may conclude that since one execution has the requested behavior then so do all executions.

Here we present some definitions in addition to a theorem that we use in our proof in Chapter 5.

Definition 2.2.3 *A process P is determinate if, for every derivative Q of P and for all alpha $\in$ Act, whenever $Q \xrightarrow{\alpha} Q'$ and $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ then $Q' \approx Q''$.*

Definition 2.2.4 *A CCS process P is confluent if it is determinate and, for each of its derivatives Q and distinct actions α, β, if $Q \xrightarrow{\alpha} Q_1$ and $Q \stackrel{\beta}{\Longrightarrow} Q_2$ then there exist Q'_1, Q'_2 such that $Q_1 \stackrel{\hat{\beta}}{\Longrightarrow} Q'_1$, $Q_2 \stackrel{\hat{\alpha}}{\Longrightarrow} Q'_2$ and $Q'_1 \sim Q'_2$.*

Similarly, for input actions that are always enabled

Definition 2.2.5 *A process P is i-confluent if for each of its derivatives Q and for all $\alpha \in L$, if $Q \xrightarrow{\alpha(v)} Q_1$ and $Q \stackrel{\alpha(u)}{\Longrightarrow} Q_2$ then there exist Q'_1, Q'_2 such that $Q_1 \stackrel{\alpha(u)}{\Longrightarrow} Q'_1$, $Q_2 \stackrel{\alpha(v)}{\Longrightarrow} Q'_2$ and $Q'_1 \sim Q'_2$.*

Theorem 2.2.6 *Suppose $P = (P_1 \mid \ldots \mid P_n)\backslash L$, where (1) each P_j is confluent, (2) each P_j is $i-confluent$ and (3) $fic(P_j) \cap fic(P_k) = \emptyset$, for all $j \neq k$. Then P is confluent.*

where $fic(P)$ are the free channels of P which are used in input position within P.

This theorem is a special case of the Theorem 3.9 of [49] and describes the conditions under which confluence is derived from the input-enabled property of a process.

The notion of confluence has also been studied in the context of other process calculi, including value-passing CCS, the π-calculus and an asynchronous version of value-passing CCS.

2. Model Checking Temporal logic is a special category of modal logic that is used for system modeling and verification, and it can express properties regarding proposition satisfaction qualified in terms of time. Temporal logic has the ability to reason about a time line. Linear time logics as LTL (Linear Temporal Logic) are restricted to this type of reasoning while branching logics as CTL (Computation Tree Logic) can reason about multiple time lines.

Model checking is a verification technique aimed at determining whether a system specification satisfies a property typically expressed as a temporal logic formula. All of the process algebras considered above are associated with well-developed temporal logics and their theories. Most of these logics are extensions of HML (Hennesy-Milner Logic) [23]. HML is a temporal logic that allows specification of modalities appropriate for checking process-algebraic specifications. In addition, versions of the logics have been proposed such that two processes are (strongly or weakly) bisimilar if and only if they satisfy the same HML properties.

3. Automated Verification Tools During the last two decades research carried out in the area of formal methods for system design and analysis has made significant success towards the development of formal methodologies for modeling systems and reasoning about their behavior. An important result of this work is the implementation of practical automated tools for verifying system properties at the push of a button. The use of such tools has already been adopted in the industry and has been applied for the analysis of large-scale real-life systems. Automated verification tools were initially developed to reason about the logical correctness of discrete state systems, but have since been extended to deal with real time and limited forms of hybrid systems. This section is concluded with a brief mention of automated tools geared towards the analysis of process-algebraic formalisms.

- *The Concurrency Workbench*

  The Concurrency Workbench is an automated tool that verifies concurrent systems expressed in the CCS process algebra. It provides users with a number of different techniques for specification and verification of finite-state concurrent systems, like equivalence checking for strong and weak bisimulations and model checking of temporal logic properties expressed in HML and CTL-.

- *The Mobility Workbench*

  The Mobility Workbench (MWB) is a tool for analyzing mobile concurrent systems described in the π-calculus. It provides implemented algorithms for checking both strong and weak equivalences as well as commands for finding deadlocks and interactively simulating an agent.

Chapter 3

Ad hoc networks

Ad hoc networks [47] are a promising technology that seeks to give solutions to areas with needs for easily deployed networking and short-lived usage. Such areas include health, science, communications, business and army.

This chapter contains a list of the key characteristics of ad hoc networks that seem to influence their behavior and a list of problems arising therein that attracted the attention of the research community. Moreover, there is a list of criteria that were used in the selection process of the algorithm for verification. At the end of the chapter, the selected algorithm is presented.

3.1 Ad hoc network characteristics

An ad hoc network is a set of autonomous nodes that configure an arbitrary topology and communicate in the absence of fixed infrastructure or centralized entity. Usually such a network consists of a set of mobile nodes that communicate with each other through a wired or wireless medium. A node can have direct communication with nodes its neighbors (in case of wireless communication its neighbors are the nodes in its transmission range) and indirect communication with the whole network (through multihop routing). Ad hoc networks usually suffer from limitations in resources like energy, computation power and storage capacity in addition to low bandwidth and low reliability (large amount of errors and collisions) in case of wireless medium.

In general ad hoc networks are characterized by:

- Distribution: Network nodes act autonomously, taking individual decisions based on local information.
- Concurrency: Network nodes act in parallel, independently of each other.
- Heterogeneity: Nodes may have different computation, storage and communication capabilities.
- Asynchronous communication: Nodes do not have to synchronize in order to communicate with each other.
- Mobility/Dynamicity: Nodes have the ability to move, changing network's topology, connectivity and size dynamically. Nodes may also fail.

More details about ad hoc networks features can be found in [47] and [12].

3.2 Problem list

The dynamic nature of ad hoc networks adds a new dimension to networking, increasing the complexity of their problems. This causes many problems that have been solved on wired networks, to be still open on ad hoc networks. Moreover, new problems need to be addressed, relating to the new features of ad hoc networks. Here, we list the problems that have been studied most widely in the literature. Some of these are directly derived from ad hoc networks characteristics, while others are used as building stones for providing solutions to more general problems.

1. **Clustering**. The problem of clustering is the problem of partitioning the network into groups, each having a centralized entity to coordinate it. Clustering ad hoc networks helps in providing energy-efficient communication between nodes. Some clustering methods are proposed in [51] and [54].
2. **Leader Election**. Ad hoc networks have no centralized entity to coordinate network nodes. But coordination is essential for many applications. Thus, the election of a leader is an important problem of ad hoc networks. In [22] the authors study the problem of leader election among other fundamental problems, while in [34, 41] the authors study this problem in special cases of ad hoc networks.

3. **Location Discovery**. Local information is always necessary in a distributed, dynamic network such as ad hoc networks. Information about location can be used to determine network topology, node location and node surroundings. In [61] the authors discuss the importance of location awareness while in [28] the authors present two algorithms for localized topology control. Two other papers that study topology control in ad hoc networks are [64] and [52].

4. **Medium Access Control - Collision Avoidance**. As said previously, the nodes in an ad hoc network communicate mostly through a wireless medium. But, the wireless medium has a serious limitation: it cannot carry more than one messages at the same time. If more than one messages are forwarded in the wireless medium then a collision occurs and all the messages on the medium are collapsed. Thus, the decision of which node will use the wireless medium to send data, when the medium is idle, is called the medium access control problem, while the matter of when the data must be sent is what the collision avoidance mechanisms try to solve. These decisions have to be taken in a distributed way since there is no centralized entity in the network. In [50] an algorithm for bandwidth adaptation allocation is presented. In [19] the authors study a subproblem of the access control problem, called the hidden terminal problem, and provide solutions to this. The hidden terminal problem appears when two nodes try to communicate with a third one, ignoring the existence of each other. This results to collision of the communication packets on the third node.

5. **Minimum Spanning Tree**. Computing a minimum spanning tree is a very important problem in networks in general, since the properties of such trees help in providing solutions to other problems such as routing, multicasting, leader election etc. One algorithm that computes a minimum spanning tree is proposed in [29], while in [4] the authors study the minimum spanning tree problem and its applicability to other fundamental problems.

6. **Power control/ Power conservation**. Since mobile nodes mostly depend on battery power, it is important to minimize their energy consumption in order to maximize the network's lifetime. Some strategies for achieving power control are proposed in [17] and [60].

7. **Quality of Service - Fairness**. As in every network, there are needs for quality of service (QoS) and fairness to the applications in the network. In ad hoc networks it is more difficult to provide quality of service because of the

low transfer rate in the wireless medium. Algorithms for QoS and fairness are proposed in [2, 59, 58].

8. **Resource allocation and Mutual Exclusion**. In an ad hoc network, nodes share resources like memory and the wireless medium. Many resources cannot be allocated to more than one node at the same time and the wireless medium is the most representative example of such a resource. In [65] and [57] the authors propose some approaches for the resource allocation problem and especially for the medium access problem.

   The problem of mutual exclusion, that is the problem of sharing a critical resource, is a special case of the resource allocation problem. A survey on mutual exclusion algorithms is presented in [5], while in [7] and [63] the authors present interesting algorithms for the mutual exclusion problem.

9. **Routing**. The routing problem is a critical problem in networks and concerns route discovery among a pair of nodes of the network and data transferring from one node to the other. Without a routing algorithm, communication between non neighboring nodes is not possible. In traditional networks, routing is handled by specific nodes, named routers. In contrast, in ad hoc networks there are no special nodes for routing, so each node has to play the role of the router in order to achieve communication through the network. Thus, the known routing algorithms that have been developed for wired networks cannot be used in ad hoc networks.

   Many new algorithms have been proposed for unicast, multicast and broadcast routing that consider the special characteristics of ad hoc networks. The most important protocols that have been developed for routing are described in [48, 53, 12] while the protocols DSDV proposed in [48] and AODV proposed in [46] are the most popular.

   In addition to the above protocols, some multipath routing protocols appeared motivated by the fact that in ad hoc networks a route can easily fail due to node movement and the sensitivity of the wireless medium. The idea is to create multiple paths between nodes in order to succeed small switching time to a new path if the used path fails. Most multipath algorithms were built over a unipath protocol as those mentioned above. Some of the most important multipath protocols were proposed in [36, 66, 44].

10. **Security**. Just like every wireless network, ad hoc networks face lots of security threats. Thus security is a problem that has attracted a lot of attention

from the research community and has to be solved in order to secure the continuous and reliable functioning of the network. In order to provide security to the network, it must be provided to every communication activity of the nodes including routing [67] and clustering [6].

3.3 Criteria

The literature provides us with many algorithms related to ad-hoc-networks problems. For this work, an algorithm was chosen to be verified by formal methods. Here, there is a list of the criteria according to which the algorithm was selected.

1. **Degree of dependency on a topology**. The first criterion the algorithm is examined for, is the degree of its dependency on network topology. Since ad hoc networks are dynamic systems, topology is continuously evolving. Thus, the degree of topology dependency of an algorithm shows whether it can be used in static, partially dynamic or fully dynamic environments. Four categories of dependency degree are listed below:

   (a) High topology dependency: The execution of the algorithms in this category is highly dependent on the topology of the network. This means, that the algorithms request full topology knowledge and they assume that the network is static. If any change in the topology occurs, the algorithm fails.

   (b) Medium topology dependency: In this category, algorithms still assume static environment, but they are able to handle node or link failures.

   (c) Low topology dependency: The execution of the algorithms in this category does not depend on network topology but there is the demand for the network always to be connected. This means, that the algorithms can be applied in dynamic systems with mobile nodes as long as the network stays connected. If the network is partitioned, then the algorithms will not be able to operate correctly.

   (d) No topology dependency: Algorithms are fully abstracted from topology issues, thus they can handle every dynamic behavior of the network, including losses, mobility and partitions.

   The dynamic nature of ad hoc networks includes losses, mobility and partitions, thus an algorithm with no topology dependency, or at least with low topology dependency would be appropriate for the sequel of this work.

2. **Fault-Tolerance**. The second criterion is whether the algorithm can handle failures. Failures vary from node and link crashes, to crashes and recoveries as well as to messages losses. Nodes may also exhibit malicious behavior. In ad hoc networks, failures are very likely to occur, thus a fault-tolerant algorithm is required.

3. **Synchronization demands**. The third criterion we consider is that of synchronization demands, based on in which algorithms are classified in three categories:

   (a) Synchronous: The algorithm demands nodes to be synchronized in order to achieve communication.

   (b) Asynchronous: Nodes do not have to synchronize in order to communicate.

   (c) Partially synchronous: Some timing issues are considered in a "loose" manner.

   Ad hoc networks are either partially synchronous or asynchronous systems so algorithms with synchronous behavior are not suitable for this work.

4. **Extended applicability**. Extended applicability is the fourth criterion which describes the capability of an algorithm to be applied in a large class of systems. Assumptions that concern network's connectivity, properties about links' possible status and messages' reliable transfer, properties about nodes' status and actions or even assumptions about timing and synchronization issues, are made by algorithms in order to assure their correct functioning. These assumptions reduce the applicability of the algorithms in the sense that, the less restrictive the assumptions are, the more wide is the class of networks that the algorithm is applicable to.

3.4 Selected Algorithm

From the problems and algorithms referred to in Section 3.2, the algorithm that was selected to be verified for the sequel of this work is named "Ad hoc On-demand Multipath Distance Vector Routing", or AOMDV for short, and it is a multipath routing algorithm for ad hoc networks. It was proposed by M.K. Marina and S.R. Das in [35, 36] and it was built over the unipath routing protocol "Ad hoc On demand Distance Vector Routing (AODV)" [46]. AOMDV inherits from the AODV

protocol its *on-demand* nature and the mechanisms for constructing and maintaining loop-free paths for a pair of source-destination nodes. It extends AODV to discover multiple paths between source and destination which are guaranteed to be loop-free and (node or link) disjoint. In this work, we examine the AOMDV version for link disjoint paths. AOMDV is interested for paths that are link disjoint respectively to a pair of nodes and not globally in the whole network. That is, a pair of paths for different source or destination nodes can share a link but a pair of paths for the same source and destination nodes should have different links.

We see that the AOMDV algorithm fulfills the criteria set in Subsection 3.3 to a great extent:

1. Degree of dependency on a topology: AOMDV has low topology dependency, because its execution does not depend on network topology but only on the local knowledge of the set of neighbors. Moreover, for the right operation of the algorithm, there is the demand that the source and the destination belong to the same partition of the network, otherwise it is not possible to create paths between them.

2. Fault-Tolerance: AOMDV inherits from AODV its fault-tolerant mechanisms like the error messages that are forwarded when a link failure is detected. Moreover, by maintaining multiple disjoint paths, in AOMDV, it is easy to change the used path to a new one in case of a failure.

3. Synchronization demands: AOMDV is an asynchronous algorithm because nodes communicate in an asynchronous manner.

4. Extended applicability: The assumptions made by AOMDV are the existence of a unique identifier for every node in the network and that all links of the network are bidirectional. The first assumption is a typical assumption of almost every ad hoc routing protocol. The second assumption is satisfied in ad hoc networks where all nodes have equal transmission range. Thus AOMDV can be applied to a relatively broad class of ad hoc networks.

AOMDV, as a routing protocol, has three major responsibilities, named *Route Discovery*, *Route Maintenance* and *Packet Forwarding*. The route discovery process concerns the creation of loop free and link disjoint paths when needed. Route Maintenance is responsible for deleting the routes that are no longer in use, to determine and report failures in the paths and, in case of a failure, to create alternate paths if possible. Packet Forwarding involves the selection of the neighbor to which the packet must be forwarded in order to reach its destination.

This work concerns the study of the Route Discovery process of AOMDV, thus the details of the other two parts of the algorithm are not presented here.

The route discovery process begins when a node, the source, needs a path to another node, the destination. If the source knows a valid path to the destination, then this path is used and the process is finished. Otherwise, it sends a route request message (RREQ) to its neighbors, starting a flooding which will reach to the destination or any other node that knows a valid path for the destination.

When an intermediate node receives a copy of this request message, it has to decide whether this copy came through a new path from the source. This path is called a "reverse path" to the source and it is saved in order to forward the replies of this route discovery process back to the source. Thus, it has to be loop free and link disjoint to any other routes to the source. If the new path is not loop free or link disjoint to the other routes, then it is discarded. Otherwise, it is accepted and inserted in the routing table. Then, the intermediate node forwards the request message if it does not know a valid path to the destination and the RREQ message was the first one received by the intermediate node.

When the request reaches the destination or any intermediate node that knows a valid path to the destination, then instead of forwarding the request, a route reply message (RREP) is sent back through a reverse path. Each intermediate node receiving a copy of the reply message, accepts the route only if it satisfies the conditions of the algorithm that ensure the loop freedom and link disjointness of this route to any other route for this destination. If the route is accepted, then the RREP message is forwarded to the source through a reverse path that was not used previously to send a copy of this reply message. As the RREP messages proceed toward the source, they establish forward loop free and link disjoint paths to the destination.

Sequence Numbers. In order to distinguish the messages for different route discovery processes, AOMDV uses *sequence numbers*. Each node keeps its own sequence number. When a node initiates a route discovery process, it increases its sequence number and includes the new number in the RREQ message. Thus the combination of the source id number, the source sequence number and the destination id number uniquely identifies the request messages of a route discovery flooding. In this way a larger sequence number indicates a newer route discovery process and it can be used to determine if a RREQ message that is received by a node is the first such message received for the specific route discovery process.

In reply messages (RREP), the sequence number of the destination is used for

the same purpose instead of the source sequence number. The sequence number of a destination node is increased only when the first RREQ message is received for the specific route discovery process.

Advertised hop count. A node in AOMDV maintains more that one paths to a destination node. These paths might have different hop counts. It is essential for the node to advertise only one hop count for all the paths that it maintains to a destination. This hop count is named *advertised hop count* and it plays a key role in ensuring loop freedom of paths. The advertised hop count for a destination is the maximum hop count of all the paths that are maintained by the node for the specific destination. When the first copy of a RREQ or a RREP is received by a node, the advertised hop count is initialized to infinity in order to allow more paths to be accepted by the node. When the first RREQ or RREP is forwarded regarding these paths, then the advertised hop count is set to the maximum hop count of the paths. From this point the advertised hop count remains the same and only paths with hop count less that the advertised hop count are accepted, thus ensuring loop freedom of the paths. The advertised hop count changes only when a new route discovery process reaches the node regarding the specific destination.

Next and last hops. Besides maintaining multiple loop free paths, AOMDV seeks to find link disjoint alternate paths. From the perspective of AOMDV algorithm, disjointness is limited to one pair of nodes and does not consider disjointness across different node pairs. This means that paths for different pairs of source and destination nodes might use the same link, but alternate paths for the same source and destination have to use different links.

To succeed link disjointness between paths of the same pair of nodes, AOMDV uses a mechanism that maintains the *last hop* information for each path in addition to the *next hop*. Next hop of a node I at a path p is the node from which I receives the message that concerns p, while last hop is the neighbor of the destination of the path. In the case of RREQ message, last hop is the neighbor of the source of the message (the source is seen as the destination of the path created by RREQ) and in case of RREP message, last hop is the neighbor of the destination. The idea behind the mechanism is that, if two paths from a node to a destination have different next hops and last hops then they are link disjoint paths. This can be ensured only if every node enforces the link disjointness property of all its paths to a specific destination. Thus a node can decide if two paths are link disjoint only by checking their next and their last hop.

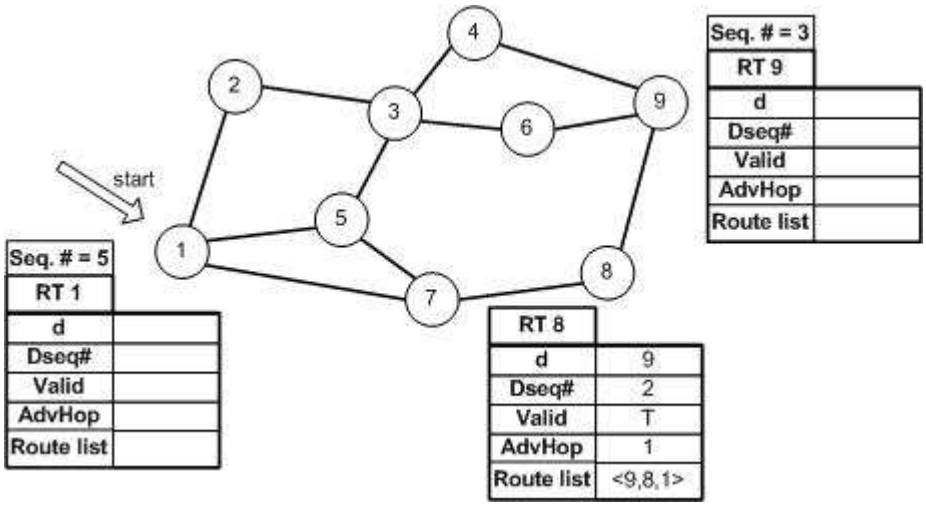

Figure 3.1: The source node 1 starts a route discovery process for destination node 9. All nodes have empty Routing tables except node 8 that have information about node 9.

AOMDV execution. The following figures show the execution of AOMDV algorithm. In Fig.3.1 we can see a network topology. In our scenario, all nodes have empty routing tables except of node with number 8 that knows a path to node with number 9. In our scenario, let the node with number 1 be the source node that requests a path to node number 9, the destination node.

In Fig.3.2 the source node 1 sends request messages to all its neighbors for the destination node 9. Each of them receiving the request message, creates a reverse path to the source and since they don't know any route to the destination node 9 they forward the message to their neighbors as shown in Figure 3.3. When the request is received by a node that knows a path to the destination node, in our scenario this is node 8, then this node replies back to the source, through the reverse path, the information it knows. This is illustrated in Figure 3.4.

When the destination receives the request messages, it sets the reverse paths to the source as any other node as in Figure 3.5. It then replies to each of the received request messages, even if it did not accept the path from that request (Figure 3.7). As the reply messages move to the source, the nodes create forward paths to the destination and accept the paths that have lower hop count that the known paths and different *next* and *last* hops (Figure 3.8). In this way, the paths that are created are loop free and link disjoint.

When the source accepts the first path to the destination, it determines that the route discovery process has finished even though more reply messages are still in their way back to the source (Figure 3.6). Finally, when more messages arrive,

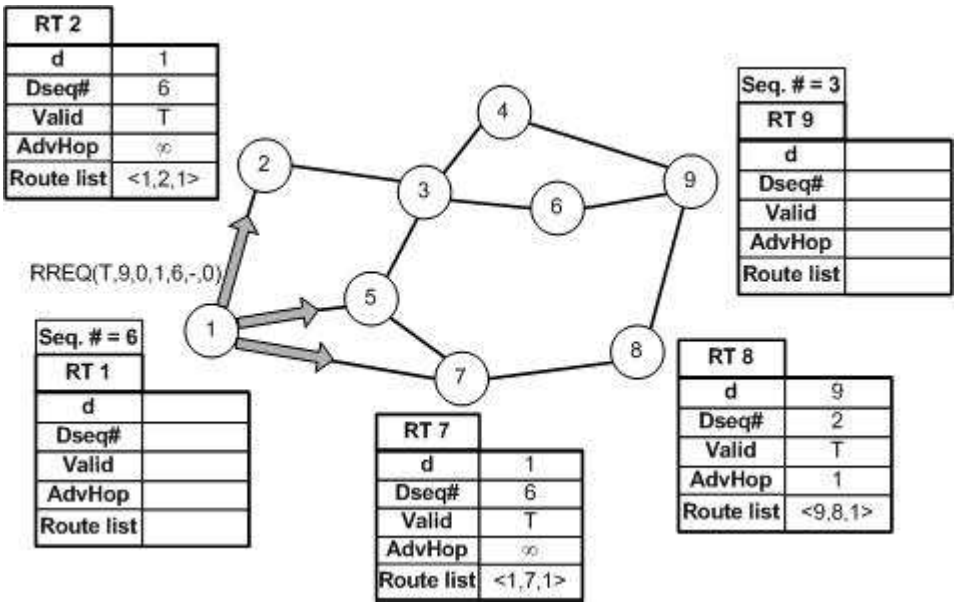

Figure 3.2: The source node 1 sends request messages to all its neighbors for the destination node 9. The nodes 2,5 and 7 create reverse paths to the source from the information of the message.

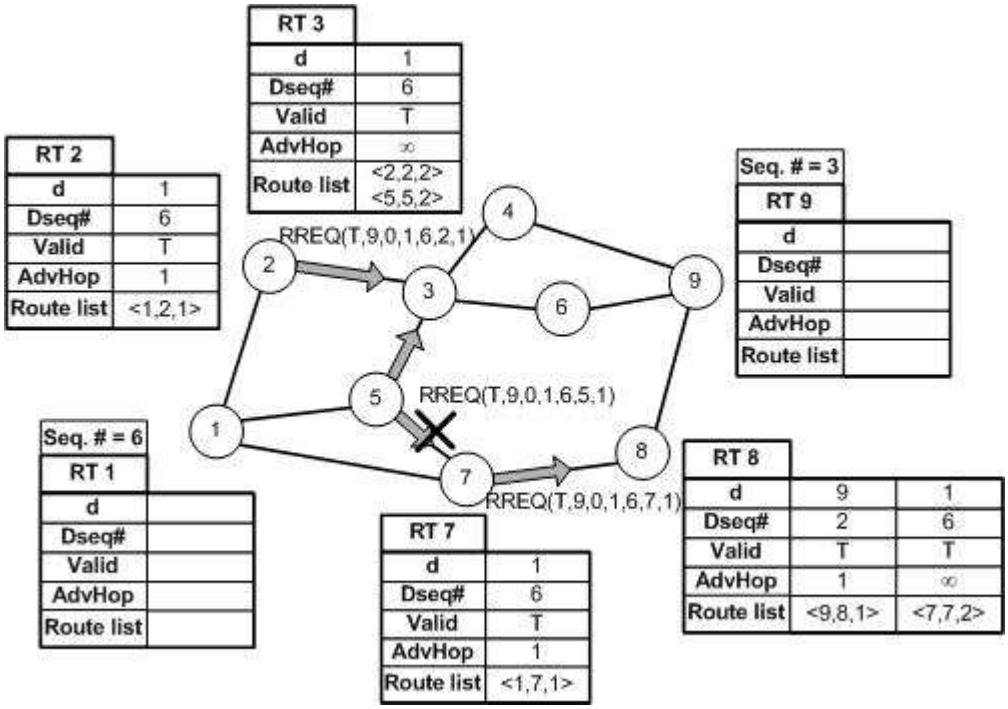

Figure 3.3: Nodes 2,5 and 7 forward the request messages to their neighbors since they don't know any path to the destination. Node 3 accepts both routes to the source since they are came from different *last* nodes. Node 7 discards the route from node 3 since it has greater hop count from the advertised hop count of the route it knows.

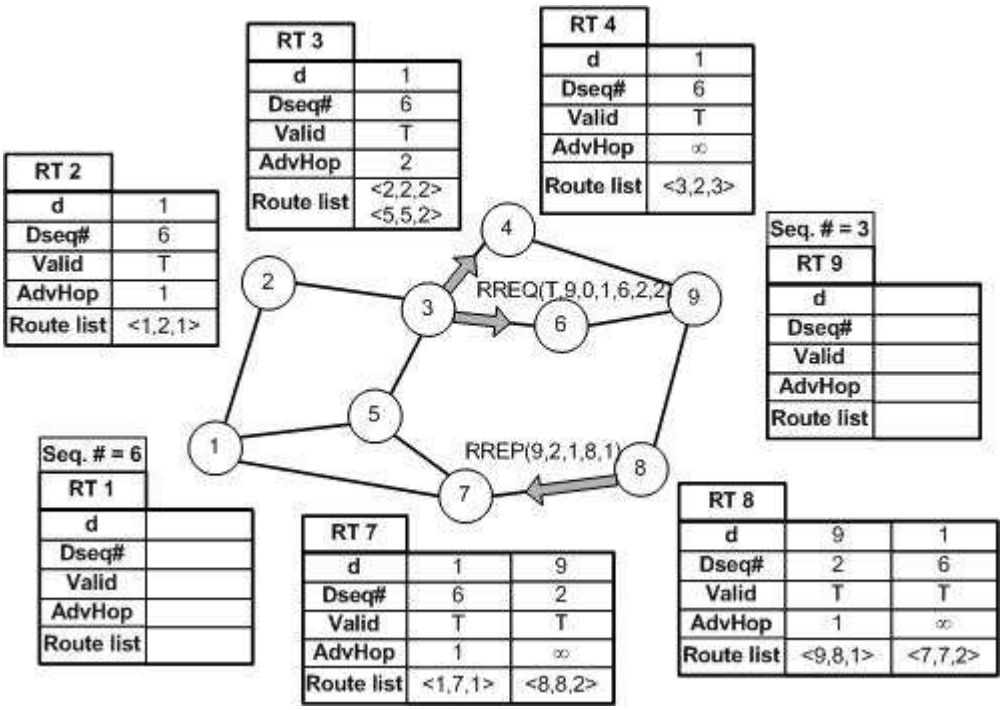

Figure 3.4: Node 3 forwards the first one of the two request messages that it accepted, in our scenario this is the message came from node 2. Node 8, since it knows a path to node 9, replies back through the reverse path it has created. Upon receiving the reply message, node 7 builds a forward path to node 9.

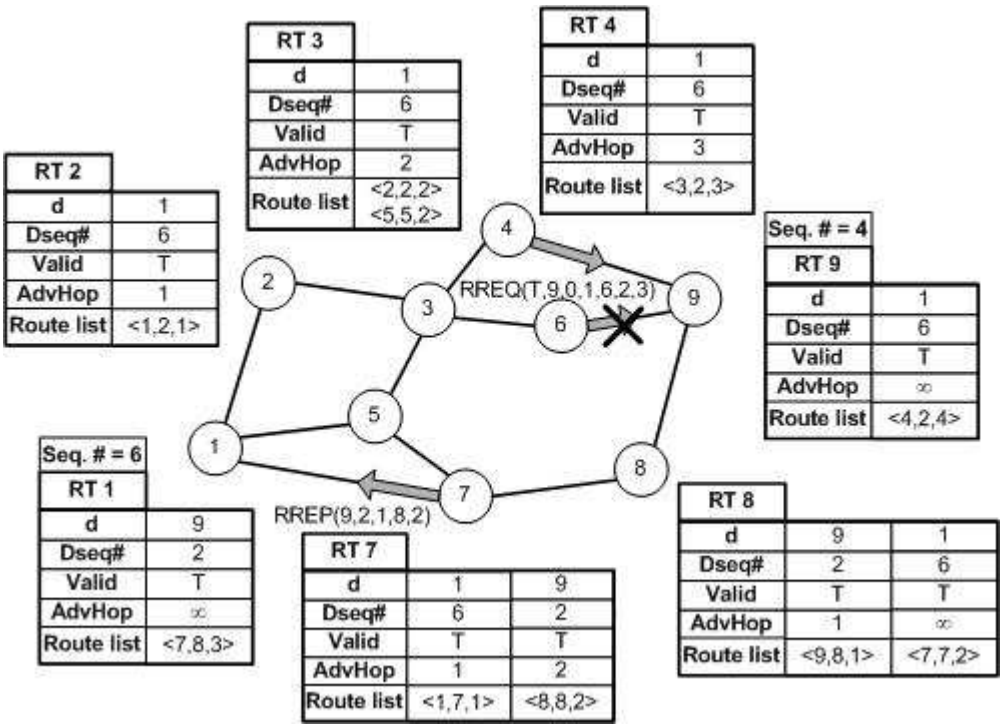

Figure 3.5: Nodes 4 and 6 send their request messages to node 9 which is the destination. Since these two messages have the same *last* hop, node 9 discards the second message that it receives, in our scenario this is the message received from node 6. Node 7, forwards the reply message to the source. Node 1 receives the reply and builds the path to node 9.

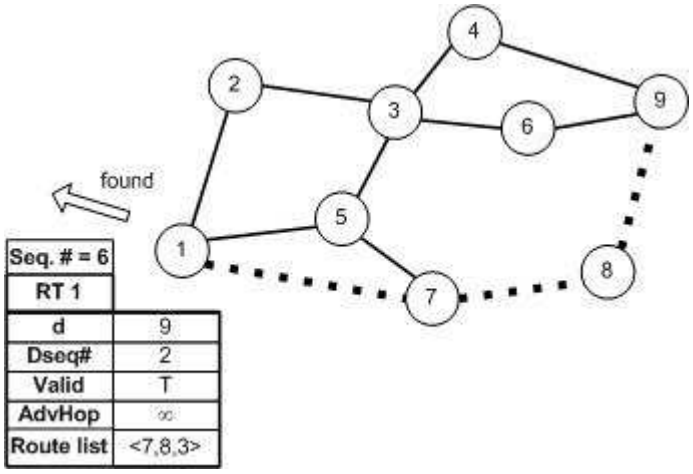

Figure 3.6: The source node, node 1 determines that the route discovery process has finished since one path has been found. With dotted line is presented the path that the source node knows so far.

it keeps the newest information (the messages with largest destination sequence number) and the paths that are loop free and link disjoint (Figure 3.8).

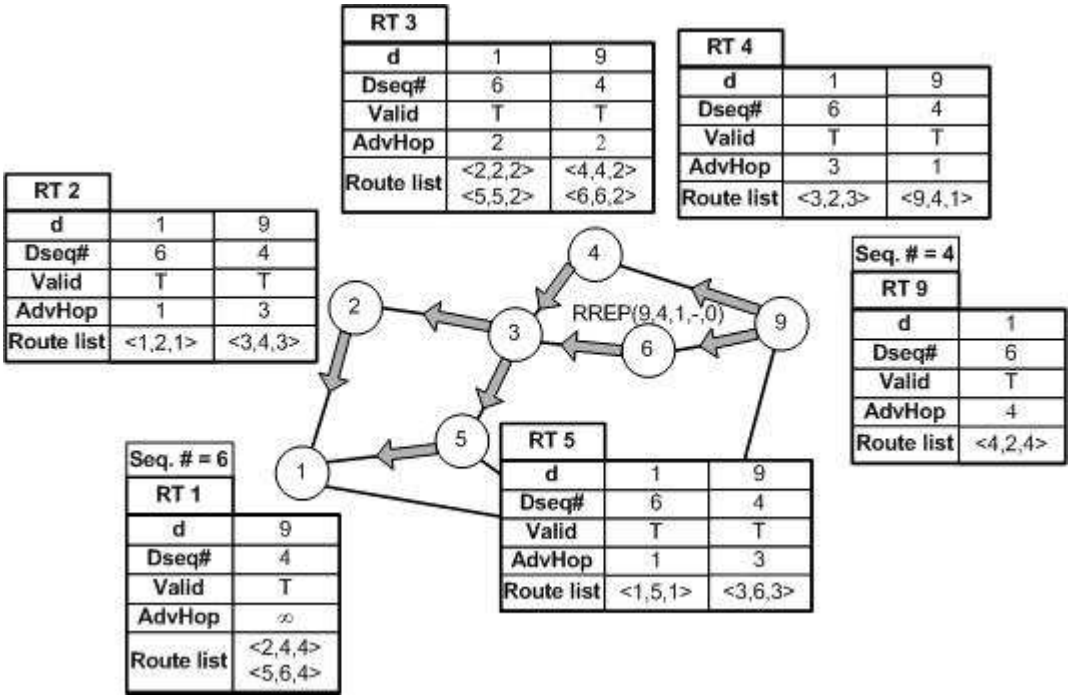

Figure 3.7: The reply messages continue their way back to the source building the forward paths to the destination. Each node accepts the paths that have grater destination sequence number, lower hop count that the known paths and different *next* and *last* hops.

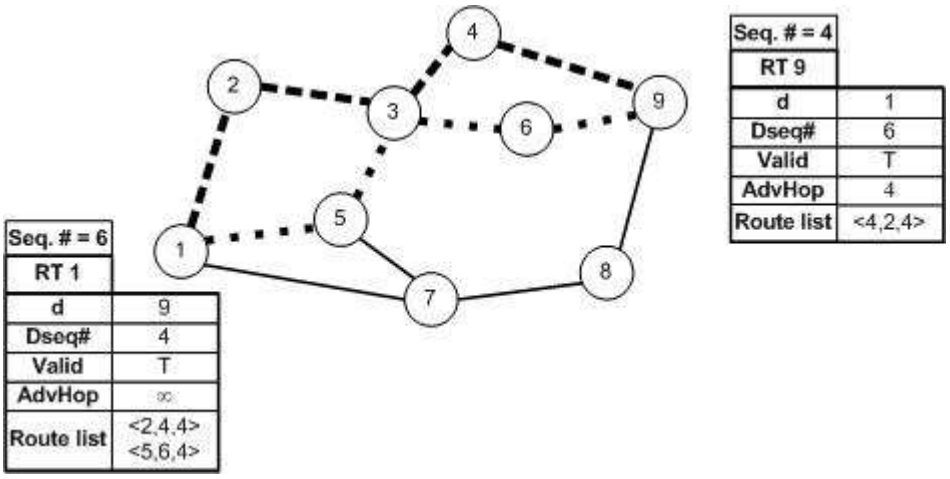

Figure 3.8: In this figure are presented the two loop free and link disjoint paths that are finally accepted from the source. The one path is presented by dashed line and the second by dotted line.

Chapter 4

Specification and Verification of AOMDV using I/O Automata

In this chapter we will apply I/O Automata for the specification and verification of AOMDV algorithm.

4.1 Specification of AOMDV

The specification of AOMDV is the composition of the Process Automata P_i and Channel Automata $C_{i,j}$, $\forall\, i, j \in I$. The specification of each automaton is given below.

Data types and Identifiers:

I: total ordered set of processes' identifiers
M: messages
$m = \langle type, data \rangle \in M$, where $type \in \{RREQ, RREP\}$;
if $type = RREQ$ then $data = \langle flag, destId, destSeqNum, srcId, srcSeqNum, last, hops \rangle$,
if $type = RREP$ then $data = \langle destId, destSeqNum, srcId, last, hops \rangle$, where
$flag$: Boolean; $destId, srcId, last \in I$; $destSeqNum, srcSeqNum, hops \in \mathbb{Z}^+$
ROUTING_TABLE: set of record tuples
$rec = \langle destId, destSeqNum, Valid, advHop, routeList \rangle \in$ ROUTING_TABLE,
where $destId \in I; destSeqNum, advHop \in \mathbb{Z}^+; Valid$: Boolean; $routeList$: list of routes;

$rt = \langle next, last, hops \rangle \in routeList$, where $next, last \in I$; $hops \in \mathbb{Z}^+$
REPLY_LIST: list of advertised routes
$rp = \langle srcId, destId, first, last \rangle \in$ REPLY_LIST
where $srcId, destId, first, last \in I$
$i, j, d \in I$
$rec_s, rec_d \in$ ROUTING_TABLE
$rt_s \in rec_s.routeList$; $rt_d \in rec_d.routeList$

Process Automaton P_i

Signature:

Input:
$receive(m)_{j,i}$
$startSearch(d)_i$
Internal:
$updateHopCount(m)_i$
Output:
$send(m)_{i,j}$
$foundAnnouncement(d)_i$

States:

$seqNum_i \in \mathbb{Z}^+$, initially 0
$replyList_i \in$ REPLY_LIST, initially *null*
$RT_i \in$ ROUTING_TABLE, initially $\emptyset$
$Neigh_i \in 2^I$: Neighbors of i, initially a fix set
$found_i \in 2^I$: destinations found, initially $\emptyset$
tosend, a vector of sets of messages, initially $tosend[j] = \emptyset \; \forall \; j \in I$
$pathAcceptance_i$: Boolean, initially *false*
$entryInit_i$: Boolean, initially *false*
$templast_i \in I$, initially $\perp$

Transitions:

input $startSearch(d)_i$
Effect:
if $\exists \; rec_d \in RT$ s.t. $rec_d.destId = d$ **then**
if $rec_d.Valid = true$ **then**
$found := found \cup \{d\}$
else
$seqNum_i := seqNum_i$+1
for all $k \in Neigh_i$ **do**
insert $\langle RREQ, \langle false, d, rec_d.destSeqNum, i, seqNum, \perp, 0 \rangle \rangle$ to $tosend_i[k]$

od
fi
else
$seqNum_i := seqNum_i{+}1$
for all $k \in Neigh_i$ **do**
insert $\langle RREQ, \langle true, d, 0, i, seqNum, \perp, 0\rangle\rangle$ to $tosend_i[k]$
od
fi

input $receive(m)_{j,i}$
Effect:
$pathAcceptance := false$
$entryInit := false$
if $m.last = \perp$ **then**
$templast = i$
else
$templast = m.last$
fi
if $m.type = RREQ$ **then**
if $\exists\ rec_s \in RT$ s.t. $rec_s.destId = m.srcid$ **then**
if $rec_s.destSeqNum < m.srcSeqNum$ **then**
$rec_s.destSeqNum := m.srcSeqNum$
$rec_s.Valid := true$
$rec_s.advHop := \infty$
$rec_s.routeList := \langle j, templast, m.hops + 1\rangle$
$pathAcceptance := true$
$entryInit := true$
elseif $rec_s.destSeqNum = m.srcSeqNum \wedge rec_s.advHop > m.hops$
$\wedge\ \ \nexists rt_s \in rec_s.routeList$ s.t. $(rt_s.next = j \vee rt_s.last = templast)$ **then**
insert $\langle j, templast, m.hops + 1\rangle$ in $rec_s.routeList$
$pathAcceptance := true$
fi
else `% if no route is known to the source`
insert $rec_s = \langle m.srcId, m.srcSeqNum, T, \infty, \langle j, templast, m.hops + 1\rangle\rangle$ in RT_i
$pathAcceptance := true$
$entryInit := true$
fi
if $i = m.destId$ **then**
if $entryInit = true$ **then**
insert $\langle RREP, \langle i, seqNum + 1, m.srcId, \perp, 0\rangle\rangle$ to $tosend_i[j]$
$seqNum := seqNum{+}1$
else
insert $\langle RREP, \langle i, seqNum, m.srcId, \perp, 0\rangle\rangle$ to $tosend_i[j]$
fi

```
    % if i is an intermediate node
  elseif pathAcceptance = true ∧ i ≠ m.destId then
   if ∃ rec_d ∈ RT s.t. rec_d.destId = m.destId then
    if rec_d.Valid = true ∧ ∃ rt_d ∈ rec_d.routeList s.t
     ⟨m.srcId, rec_d.destId, ∗, rt_d.last⟩ ∉ replyList then
      insert ⟨RREP, ⟨m.destId, rec_d.destSeqNum, m.srcId, rt_d.last, rec_d.advHop⟩⟩
        on tosend_i[j]
      insert ⟨m.srcId, m.destId, templast, rt_d.last⟩ in replyList
    elseif entryInit = true ∧ rec_d.Valid = false then
     for all k ∈ Neigh_i do
      insert ⟨RREQ, ⟨false, m.destId, max(m.destSeqNum, rec_d.destSeqNum),
       m.srcId, m.srcSeqNum, templast, rec_s.advHop⟩⟩ to tosend_i[k]
     od
    fi
   elseif entryInit = true then        % if no route is known to the destination
    for all k ∈ Neigh_i do
     insert ⟨RREQ, ⟨true, m.destId, 0, m.srcId, m.srcSeqNum, templast,
      rec_s.advHop⟩⟩ to tosend_i[k]
    od
   fi
  fi        % end of m.type = RREQ

 elseif m.type = RREP then
  if ∃ rec_d ∈ RT s.t rec_d.destId = m.destId then
   if rec_d.destSeqNum < m.destSeqNum then
    rec_d.destSeqNum := m.destSeqNum
    rec_d.Valid := true
    rec_d.advHop := ∞
    rec_d.routeList := ⟨j, templast, m.hops + 1⟩
    pathAcceptance := true
    entryInit := true
   elseif rec_d.destSeqNum = m.destSeqNum ∧ rec_d.advHop > m.hops then
    if ∄ rt_d ∈ rec_d.routeList s.t. rt_d.next = j ∨ rt_d.last = templast then
     insert ⟨j, templast, m.hops + 1⟩ in rec_d.routeList
     pathAcceptance := true
    fi
   fi
  else        % if no route is known to the destination
   insert ⟨m.destId, m.destSeqNum, T, ∞, ⟨j, templast, m.hops + 1⟩⟩ in RT_i
   pathAcceptance := true
   entryInit := true
  fi
  if i = m.srcId ∧ entryInit = true then
   found := found ∪ {m.destId}
  if i ≠ m.srcId ∧ pathAcceptance = true then    % if i is an intermediate node
```

if $\exists\ rec_s \in RT$ s.t. $rec_s.destId = m.srcId$
$\wedge\ \exists\ rt_s \in rec_s.routeList$ s.t. $\langle m.srcId, m.destId, rt_s.last, * \rangle \notin replyList$ **then**
insert $\langle RREP, \langle m.destId, m.destSeqNum, m.srcId, templast, rec_d.advHop \rangle \rangle$
to $tosend_i[rt_s.next]$
insert $\langle m.srcId, m.destId, rt_s.last, templast \rangle$ in $replyList$
fi
fi `% end of pathAcceptance check`
fi

internal $updateHopCount(m)_i$
Precondition:
$m \in tosend[j]$, $j \in Neigh_i$
if $m.type = RREQ$ **then**
$m.data.hops \neq \max(rt.hops | \forall\ rt \in rec.routeList)$
where $rec \in RT_i$ s.t. $rec.destId = m.srcId$
elseif $m.type = RREP$ **then**
$m.data.hops \neq \max(rt.hops | \forall\ rt \in rec.routeList)$
where $rec \in RT_i$ s.t. $rec.destId = m.destId$
fi
Effect:
$m.data.hops := \max(rt.hops | \forall\ rt \in rec.routeList)$
$rec.advHop := m.data.hops$

output $send(m)_{i,j}$
Precondition:
$m \in tosend[j]$
$j \in Neigh_i$
if $m.type = RREQ$ and $\exists rec \in RT_i$ s.t. $rec.destId = m.srcId$ **then**
$m.data.hops = \max(rt.hops | \forall\ rt \in rec.routeList)$
elseif $m.type = RREP$ and $\exists rec \in RT_i$ s.t. $rec.destId = m.destId$ **then**
$m.data.hops = \max(rt.hops | \forall\ rt \in rec.routeList)$
fi
Effect:
remove m from $tosend[j]$

output $foundAnnouncement(d)_i$
Precondition:
$d \in found_i$
Effect:
$found = found - \{j\}$

Tasks:

$\{updateHopCount(m)_i : m \in M\}$
For every $j \neq i$ $\{send(m)_{i,j} : m \in M\}$
$\{foundAnnouncement(d)_i : d \in I\}$

Channel Automaton $C_{i,j}$

Signature:

Input:
$send(m)_{j,i}$
Output:
$receive(m)_{i,j}$

States:

MSG, a set of elements in M, initially $\emptyset$

Transitions:

input $send(m)_{j,i}$
Effect:
$MSG := MSG \cup \{m\}$

output $receive(m)_{i,j}$
Precondition:
$m \in MSG$
Effect:
$MSG := MSG - \{m\}$

Tasks:

$\{receive(m)_{i,j} : m \in M\}$

4.2 Correctness Proof

In this section we aim to prove that AOMDV has the desired behavior. To achieve this we use the method of $Safety$ and $Liveness$ properties, that is to prove firstly that the algorithm will never act doing some "bad" things and then to prove that something *good* eventually happens.

4.2.1 Safety Properties

We start our proof of correctness by the proof of some safety properties. Each computation, or route discovery process, starts when a node i, the source, executes a $startSearch(j)_i$ input action and ends when the same node i, executes a $foundAnnouncement(j)_i$ output action. Thus, $startSearch(j)_i$ input means that i needs a path to a destination d and $foundAnnouncement(j)_i$ output means that i knows at least one path to j. We first want to ensure that no computation ever starts without a $startSearch(j)_i$ action, in other words, every $foundAnnouncement(j)_i$ action is the result of a computation started by a $startSearch(j)_i$ action.

Lemma 4.2.1 *If a $foundAnnouncement(j)_i$ output occurs, then there must exists a preceding $startSearch(j)_k$ input where $k = i$.*

Proof. The proof is by investigation of the code. From the precondition of the $foundAnnouncement(j)_i$ action, it's obvious that for this action to occur, it must hold that $j \in found_i$, which is empty initially. The only two cases that a node is inserted in $found_i$ set are the following:

- At $startSearch(j)_i$ action when $\exists rec \in RT_i$ s.t. $rec.destId = j$ and $rec.Valid = true$. In this case $found := found \cup \{j\}$, thus a $startSearch(j)_i$ input occurs before a $foundAnnouncement(j)_i$ output.

- At $receive(m)_{l,i}$ action when $m.type = RREP$, $i = m.srcId$ and $entryInit = true$. If $m.destId = j$ then $found := found \cup \{j\}$ and the $foundAnnouncement(j)_i$ can occur. Now let us check where this message is from. Node i has received this message m from a node l. This implies a preceding $send(m)_{l,i}$ action. Node l can send m message s.t. $m.type = RREP$ in the following cases:

  - In $receive(m)_{k,l}$ action, when $m.type = RREQ$ and $l = m.destId$ (that is $l = j$).

– In $receive(m)_{k,l}$ action, when $m.type = RREQ$, $pathAcceptance = true$, $l \neq m.destId$, $\exists rec \in RT_l$ s.t. $rec.destId = m.destId$, $rec.Valid = true$ and $\exists rt \in rec.routeList$ s.t. $\langle m.srcId, rec.destId, *, rt.last\rangle \notin replyList_l$.

– In $receive(m)_{k,l}$ action, when $m.type = RREP$, $pathAcceptance = true$, $l \neq m.srcId$, $\exists rec \in RT_l$ s.t. $rec.destId = m.srcId$ and $\exists rt \in rec.routeList$ s.t. $\langle m.srcId, rec.destId, rt.last, *\rangle \notin replyList_l$.

For the last case we apply recursively the same arguments. For the first two cases we have to check when node k can send a message m s.t. $m.type = RREQ$. Node u can send message m s.t. $m.type = RREQ$ in the following cases:

– At $startSearch(j)_k$ action when $\exists rec \in RT_k$ s.t. $rec.destId = j$ but $rec.Valid = false$. In this case $m.srcId = k$ thus a $startSearch(j)_k$ input occurs before a $foundAnnouncement(j)_i$ output. Note that $k = m.srcId = i$.

– At $startSearch(j)_k$ action when $\nexists rec \in RT_k$ s.t. $rec.destId = j$. Again $m.srcId = k$ and a $startSearch(j)_k$ input occurs before a $foundAnnouncement(j)_i$ output where $k = m.srcId = i$.

– In $receive(m)_{u,k}$ action, when $m.type = RREQ$, $pathAcceptance = true$, $k \neq m.destId$, $\exists rec \in RT_l$ s.t. $rec.destId = m.srcId$, $rec.Valid = false$ and $entryInit = true$.

– In $receive(m)_{u,k}$ action, when $m.type = RREQ$, $pathAcceptance = true$, $k \neq m.destId$, $\nexists rec \in RT_l$ s.t. $rec.destId = m.srcId$ and $entryInit = true$.

For the last two cases, the proof continues recursively.

We have shown that in every case if a $foundAnnouncement(j)_i$ output occurs, then a $startSearch(j)_k$ input had occurred earlier in the execution and $k = i$. This completes the proof. □

We have shown that no computation ever starts without a $startSearch(j)_i$ action, but what about more that one answers to the same start action? Here, we show that for every $startSearch(j)_i$ input, there won't occur more than one $foundAnnouncement(j)_i$ output, if such an output occurs.

Lemma 4.2.2 *For every $startSearch(j)_i$ input, there is at most one succeeding $foundAnnouncement(j)_i$ output.*

Proof. From Lemma 4.2.1 we know that if a $foundAnnouncement(j)_i$ output occurs, then there is a preceding $startSearch(j)_i$ input. A code investigation of $foundAnnouncement(j)_i$ action shows that the effect of this action is to remove j from $found_i$ set, thus $foundAnnouncement(j)_i$ will never occur again unless a new $startSearch(j)_i$ input occurs in order to put j again in $found_i$. □

During the execution of AOMDV, each node i holds the routing information for other destinations in a routing table. For each destination, i holds a unique record in its routing table, thus, there are no duplicates in the routing information for other nodes. The alternative paths to a destination d that are maintained by i are kept as unique routes $rt \in rec.routeList$, where rec is the specific record for the destination j in i's routing table. With the next invariant we show the uniqueness of the record for each destination in i's routing table.

Invariant 1 *Given any execution of algorithm AOMDV, any reachable state s and any node $i \in I$, then $\forall\, rec_k, rec_l \in s.RT_i$, $rec_k.destId \neq rec_l.destId$.*

Proof. The proof is by induction on length of the execution. The base case is trivial since $RT_i = \emptyset$, $\forall\, i \in I$. Let the invariant hold for state s and consider step (s, π, s'). We will prove that the invariant also holds at state s'. If $\pi = startSearch(j)_i, i,j \in I$ then RT_i is not altered from the respective in s state, thus the statement holds. The same holds for $\pi = send(m)_{i,j}$ and $\pi = foundAnnouncement(j)_i$. In the case that $\pi = updateHopCount(m)_i$, $rec \in RT_i$ might be modified but $rec.destId$ is unchanged while no else rec is inserted in RT_i. Thus, $rec.destId$ is unique and statement holds. If $\pi = receive(m)_{j,i}$ then there are the following cases:

- If $\not\exists\, rec \in RT_i$ s.t. $rec.destId = j$ then a record is inserted in RT_i s.t. $rec.destId = j$. Thus the statement holds.
- If $m.type = RREQ$ and $\exists rec \in RT_i$ s.t. $rec.destId = m.srcid$ then no rec is inserted in RT_i. Instead, rec might be modified but keeps $rec.destId = m.srcid$ unchanged. Thus $rec.destId$ is unique and statement holds.
- If $m.type = RREQ$ and $\not\exists\, rec \in RT_i$ s.t. $rec.destId = m.srcid$ then a relative rec is inserted in RT_i. Again, the statement holds.
- If $m.type = RREP$ and $\exists\, rec \in RT_i$ s.t. $rec.destId = m.destId$ then no other rec is inserted in RT_i. Thus the statement holds.
- If $m.type = RREP$ and $\not\exists\, rec \in RT_i$ s.t. $rec.destId = m.destId$ then a relative rec is inserted in RT_i. Again, the statement holds.

- In any other case of this action, RT_i remains the same as in state s, thus the invariant holds.

This completes the proof. □

We continue with the proof of an invariant that sets the foundations for a node's localized decisions about rejecting paths that might end up in a cycle. Recall that no path is accepted if its hop count is greater than the advertised hop count that maintains a node for a destination. This lemma says that once the advertised hop count for a destination node is set to the maximum hop number of the known paths, then it remains the same until the destination sequence number is changed.

Lemma 4.2.3 *For any record $rec \in RT_i, i \in I$, once $rec.advHop$ gets a value for a specific $rec.destSeqNum$, it remains the same until the $rec.destSeqNum$ is changed.*

Proof. The proof is by investigation of the code. When the first message, with a new sequence number is received by a node i, the advertised hop count is set to infinity. This can be seen in two cases of $receive(m)_{j,i}$ action. The first is when $m.type = RREQ$ and either $\exists\, rec \in RT_i$ s.t. $rec.destId = m.srcId$ and $rec.destSeqNum < m.srcSeqNum$ or $\nexists\, rec \in RT_i$ s.t. $rec.destId = m.srcId$. The second case is symmetrical to the first for $m.type = RREP$, when either $\exists\, rec \in RT_i$ s.t. $rec.destId = m.destId \wedge rec.destSeqNum < m.destSeqNum$ or $\nexists\, rec \in RT_i$ s.t. $rec.destId = m.destId$. In these cases $rec.advHop := \infty$. When a message m is prepared to be sent, whether $m.type = RREQ$ or $m.type = RREP$, then m inherits the advertised hop count of the paths that advertises, that is $m.hops = rec.advHops$. It is obvious that if m is the first such message, then $m.hops = \infty$. A special case is when $i = m.destId$ where $m.hops = 0$.

By the preconditions of the $send(m)_{i,j}$ action it is obvious that the message m cannot be sent as long as $m.data.hops = \infty$. This condition trickers the internal action $updateHopCount(m)_i$ in which $m.data.hops$ and $rec.advHop$ are set equal to $\max(rt.hops \,|\, \forall rt \in rec.routeList)$.

When any other message m is received by i with the same sequence number then the path is accepted only if $m.hops < rec.advHop$. This can be seen in the following cases: When $m.type = RREQ$ and $\exists\, rec \in RT_i$ s.t. $rec.destId = m.srcId$ and $rec.destSeqNum = m.srcSeqNum$ or when $m.type = RREP$ and $\exists rec \in RT_i$ s.t. $rec.destId = m.destId \wedge rec.destSeqNum = m.destSeqNum$. By this condition, even if new paths are accepted by i, these paths have less hop count that the advertised hop count. Thus, as long as $rec.destSeqNum$ remains the same, $rec.advHops = \max(rt.hops \,|\, \forall rt \in rec.routeList)$. □

Now, we prove an Invariant that leads to the conclusion that every path created by AOMDV is loop free.

Invariant 2 *Given any execution of algorithm AOMDV, any reachable state s and any pair of nodes $i, j \in I$, if $\exists\, rec \in s.RT_i$, s.t. $rec.destId = j$ then for every $rt \in rec.routeList$, rt induces a loop free path from i to j.*

Proof. A path is loop free if every node in the path appears exactly one time. Here we prove that every node will never appear twice in a path. The proof is by induction on the states of the execution. The base case is trivial since $RT_i = \emptyset$, $\forall\, i \in I$. Let the invariant hold for state s and consider step (s, π, s'). We will prove that the invariant also holds at state s'. If $\pi = startSearch(d)_i, send(m)_{i,j}$ or $foundAnnouncement(d)_i, d \in I$ then $s'.RT_i = s.RT_i$, thus the statement holds. If $\pi = updateHopCount(m)_i$ there might be a modification of $rec \in RT_i$, but this modification is not over the paths of the record thus, the statement holds. If $\pi = receive(m)_{k,i}$ then there are the following cases:

- If $m.type = RREQ$ and $m.srcid = j$ and $\nexists\, rec \in RT_i$ s.t. $rec.destId = j$ then a record is inserted in RT_i s.t. $rec.destId = j$ and $rt.next = k$. This means that i participates for the first time on the path to j induced by rt. Moreover, by inductive hypothesis, the path was loop free until node k on state s, thus the path is loop free.

- If $m.type = RREQ$, $m.srcid = j$ and $\exists\, rec \in RT_i$ s.t. $rec.destId = j$ and $m.srcSeqNum > rec.destSeqNum$, then the record's fields except $rec.destId$ are deleted and filled with the information of the message m. Thus, every $rt \in rec.routeList$ is deleted and at state s', $rec.routeList$ contains only one rt that induce the path through k. In other words, i cannot be in the same path for the second time, since every path known so far is dropped. By inductive hypothesis, the path was loop free until node k on state s, thus the path is loop free. Moreover, note that because of Invariant 1, rec is unique in RT_i.

- If $m.type = RREQ$, $m.srcid = j$ and $\exists\, rec \in RT_i$ s.t. $rec.destId = m.srcid$, $m.srcSeqNum = rec.destSeqNum$ then there are two cases:

  1. the new path is rejected. This happens when either $m.hops > rec.advHop$ or $\exists\, rt' \in rec.routeList$ s.t. $rt'.next = k$ or $rt'.last = m.last$ (or $rt'.last = i$ if $m.last = \perp$). In this case $s'.RT_i = s.RT_i$, thus the statement holds.

2. the new path is accepted. This happens when $m.hops < rec.advHop$ and $\nexists rt' \in rec.routeList$ s.t. $rt'.next = k$ or $rt'.last = m.last$ (or $rt'.last = i$ if $m.last = \perp$). In this case i is added to the path to j as the node after k. According to the inductive hypothesis for state s, the path of k with destination j is loop free. For contradiction purpose, let us assume that i enters the path for the second time, thus in the path accepted there exists a link $(i, l), l \in I$. This link was created when a pair of $send(m')_{i,l} - receive(m')_{i,l}$ actions occurred at a state $s' < s$, where $m'.type = RREQ$ and $m'.data.srcid = j$ or $m'.type = RREP$ and $m'.data.destid = j$. Moreover $m'.data.hops$ is equal to the maximum $rt.hop$, $\forall rt \in rec.routeList$. This is enforced by the preconditions of $send(m')_i, u$ and $updateHopCount(m')_i$ actions. At the same time, $updateHopCount(m')_i$ action, enforces that $m'.data.hops = rec.advHop$. The creation of the link implies that l accepted this path.

   According to the new path that was accepted, there are two cases; the path was accepted before or after the message m' was sent. If the path was accepted before the m' was sent, then $s < s'$ and the path was accepted before the creation of the link (i, l), a contradiction. Now, if the path was accepted after m' was sent, then $m.data.hops > m'.data.hops$ since the new path contains at least one link, the link (i,l), more than the previous path that was advertised in m'. Recall that for a path to be accepted it must hold that $m.data.hops < rec.advHop$. But $rec.advHop = m'.data.hops$ and according to Lemma 4.2.3 $rec.advHop$ was not changed from s' (since $rec.destSeqNum$ did not change), which leads to a contradiction. Thus i participates in the path for the first time, so the new rt inserted in $rec.routeList$ induces a loop free path from i to j.

- If $m.type = RREP$ and $m.data.destId = j$, we have similar cases since in routing table both source and destination of the route discovery process are handled as different destinations of paths from i. This allows us to use similar arguments for each case.

This completes the proof. □

Here is the first of our main results. This result regards the loop-freedom of each path created by AOMDV in a route discovery process between a source node i and a destination node j.

Theorem 4.2.4 *For every pair of nodes $i, j \in I$ s.t. $\exists\, startSearch(j)_i$ input, if i knows a path to j, then this path is loop free.*

Proof. The proof comes straight forward from Invariant 2 since, if i knows a path to j then there exists $rec \in s.RT_i$ s.t. $rec.destId = j$ with at least one $rt \in rec.routeList$. □

We continue with an invariant that proves that any two paths known by a node i for a specific destination are link-disjoint.

Invariant 3 *Given any execution of algorithm AOMDV, any reachable state s and any pair of nodes $i, j \in I$, if $\exists\, rec \in s.RT_i$ s.t. $rec.destId = j$, then for every pair of $rt_1, rt_2 \in rec.routeList$:*

1. *$rt_1.next \neq rt_2.next$ and $rt_1.last \neq rt_2.last$ and*
2. *rt_1 and rt_2 induce link disjoint paths from i to j.*

Proof. The first part of the invariant is enforced in action $receive(m)_{k,i}, k \in I$ in the following cases:

1. $m.type = RREQ \wedge m.data.srcId = j$ when $\exists rec \in RT_i$ s.t. $rec.destId = j$, $rec.destSeqNum = m.srcSeqNum$ and $rec.advHop > m.hops$ by the condition that there exists no $rt \in rec.routeList$ s.t. $rt.next = k$ or $rt.last = m.last$ (or $rt.last = i$ if $m.last = \perp$).
2. $m.type = RREP \wedge m.data.srcId = j$ when $\exists\, rec \in RT_i$ s.t. $rec.destId = j$, $rec.destSeqNum = m.destSeqNum$ and $rec.advHop > m.hops$ by the relative condition.

The proof of the second part is an induction on the states of the execution. The base case is trivial on the initial state, since $RT_i = \emptyset,\ \forall\, i \in I$. Let the invariant (second part) hold for state s. We will prove that it also holds at state s', s.t. (s, π, s'). If $\pi = startSearch(d)_i, send(m)_{i,k}$ or $foundAnnouncement(d)_i, i, k, d \in I$ then RT_i is not altered from the respective in s state, thus the statement holds by the inductive hypothesis. If $\pi = updateHopCount(m)_i$, there might be a modification of $rec \in RT_i$, but $rec.routeList$ is unchanged thus, the statement holds. If $\pi = receive(m)_{k,i}$ then there are the following cases:

- If $m.type = RREQ, m.srcid = j$ and $\nexists\, rec \in RT_i$ s.t. $rec.destId = j$ or $\exists\, rec \in RT_i$ s.t. $rec.destId = j$ but $m.srcSeqNum > rec.destSeqNum$, then the record is initiated with one rt. The invariant is reestablished.

- If $m.type = RREQ$, $m.srcid = j$ and $\exists rec \in RT_i$ s.t. $rec.destId = m.srcid$, $m.srcSeqNum = rec.destSeqNum$ then there are two cases:

  1. the new path is rejected. This happens when either $m.hops > rec.advHop$ or $\exists rt' \in rec.routeList$ s.t. $rt'.next = k$ or $rt'.last = m.last$ (or $rt'.last = i$ if $m.last = \perp$). In this case $s'.RT_i = s.RT_i$, thus the statement holds.

  2. the new path is accepted. This happens when $m.hops < rec.advHop$ and $\nexists rt' \in rec.routeList$ s.t. $rt'.next = k$ or $rt'.last = m.last$ (or $rt'.last = i$ if $m.last = \perp$). In this case i is added to a new path to j as the node after k and a new rt (for the new path) is inserted in $rec.routeList$. Let us check the link disjointness property of the new rt (name rt_1) with any $rt_2 \in rec.routeList$. Note that rt_1 and rt_2 have different *next* and *last* nodes according to first part of this invariant. Moreover, every previous node in rt_1 and rt_2 have link-disjoint paths according to the inductive hypothesis. Thus rt_1 and rt_2 induce link disjoint paths.

- If $m.type = RREP$ and $m.data.destId = j$, we have similar cases since in routing table both source and destination of the route discovery process are handled as different destinations of paths from i. This allows us to use similar arguments for each case.

This completes the proof. □

Next we present the second of our main results. This result regards the link-disjointness of any pair of paths created by AOMDV in a route discovery process between a source node i and a destination node j.

Theorem 4.2.5 *For every pair of nodes $i, j \in I$ s.t. $\exists startSearch(j)_i$ input, if i knows paths p_1, p_2 to j, then p_1, p_2 are link disjoint paths.*

Proof. The proof comes straight forward from Invariant 3 if $\exists rec \in RT_i$ s.t. $rec.destId = j$, and $\exists rt_1, rt_2 \in rec.routeList$ that induce paths p_1 and p_2 respectively. □

4.2.2 Liveness Properties

After showing that nothing "bad" will happen in AOMDV execution, we have to show that something "good" will eventually happen. That is, the route discovery process will eventually terminate by source announcing that at least one path was

found for the desired destination. In this section we consider only fair executions of the algorithm AOMDV and we assume that messages are delivered in finite time.

We start this proof with a simple case. In this case only one source node i starts a route discovery process for a destination node j. We prove that after starting this process, eventually i announces that a path to j was found.

Lemma 4.2.6 *For a $startSearch(j)_i$ input, eventually a corresponding $foundAnnouncement(j)_i$ output occurs.*

Proof. The proof is by construction. Suppose that we start from an initial situation in which a $startSearch(j)_i$ input occurs. If i already knows a valid path to j, that is $\exists rec \in RT_i$ s.t. $rec.destId = j$ and $rec.Valid = true$, then j is inserted in set $found$ which enables the $foundAnnouncement(j)_i$ to occur. From Lemma 4.2.2 we know that this is the only $foundAnnouncement(j)_i$ action that occurs.

If i knows an invalid path to j, that is $\exists rec \in RT_i$ s.t. $rec.destId = j$ and $rec.Valid = false$, then i increase its sequence number to denote a new path discovery process and prepares a new request message with the information in rec. If i does not know any path to j, that is $\nexists rec \in RT_i$ s.t. $rec.destId = j$, then i does the same actions as in the case of invalid path, but the message is different since i has no information about j. In this case the message denotes that the destination j is unknown, that is $m.data.U = true$ and $m.data.destSeqNum = 0$. Both these cases start a sequence of send/receive actions for route discovery of at least one path from i to j. Recall that from Theorem 4.2.4 any path created is loop free and from Theorem 4.2.5, if more than one paths are discovered, then these paths are link disjoint.

The request messages will be sent to every k s.t. $k \in Neig_i$, thus at hop 1 from i. Upon receive the request message, k checks whether it knows any path to i, that is $\exists rec \in RT_k$ s.t. $rec.destId = i$. If yes then it reinitializes the record since $m.srcSeqNum > rec.destSeqNum$ (m is the first message with the specific $m.srcSeqNum$) and goes forward to check for the destination. If no then it inserts a new record with the information of the message and again goes forward to check its knowledge for the destination. In both cases k sets $pathAcceptance = true$ and $entryInit = true$.

If k is the destination, that is $k = m.destId = j$, then it prepares a message m for reply which will be send back to i. If $k \neq j$ and knows a valid route to j, that is $\exists rec \in RT_k$ s.t. $rec.destId = j$ and $rec.Valid = true$, then again it prepares a message m for reply. In this case, k selects one route from the known routes to

include in m. In any other case k forwards the request to its neighbors. These actions occur in round 1.

In the last case, every $l \in Neigh_k$ who receives this $RREQ$ for the first time, that is l is at hop 2 from i, acts at the same way as described above for k. If this message is not the first one, then it holds that $\exists rec \in RT_l$ s.t. $rec.destId = m.srcId$ and $rec.destSeqNum = m.destSeqNum$. In this case, the path is accepted only if it is a loop free path and it is link disjoint to any previously accepted path. If l accepts the path, it inserts the new information in $rec.routeList$ and sets $pathAcceptance = true$ which makes l decide its next actions according to the knowledge of destination. If l is the destination, then it prepares a message m for reply which will be send back to k. If $k \neq j$ and knows a valid route to j, then it selects one route from the known routes for destination j and includes it in the reply message m. If l does not know any valid route to j then it does not forwards the request to its neighbors since it has forwarded the first request message it has received. This process continues recursively until the message reaches the destination or a node that knows a valid path to j.

In the worst case, given the flooding property, the connectivity of the network and the above discussion, a request message will reach the destination j at D hops, where D is the maximum loop free path from i to j. When j receives the request message, then it checks the path to the source as every previous node, to determine if it should accept the path or not. If it accepts the path, then it inserts the path in its routing table with destination i, increases is sequence number if it is the first RREQ message that receives for this route discovery process and answers back through the path with a reply message. In order to create more link disjoint paths, the destination replies in every RREQ message that receives, even if it rejected the path.

Each node k that receives a reply message RREP, decides if it will accept the new path to the destination j or not. That is, k checks whether it knows any path to j, that is $\exists rec \in RT_k$ s.t. $rec.destId = j$. If yes then it checks if $m.destSeqNum > rec.destSeqNum$. In this case it reinitializes the record with the new information about the destination. Otherwise, it checks if the path is loop free and link disjoint to the paths that already knows. If it does not know any path to j, then it inserts a new record with the information of the message. If the path is accepted and k is the source, that is $k = m.srcId = i$, then j is inserted in set $found$ only if it is the first RREP that receives. Otherwise, the message is discarded. When $j \in found$ then the $foundAnnouncement(j)_i$ action is enabled. From Lemma 4.2.2 we know that no more $foundAnnouncement(j)_i$

actions will occur. If $k \neq i$ then it forwards the request through any known route to i which was not used previously to send RREP messages for the same route discovery process. In this way, the RREP messages move backwards to the source. In the worst case i will eventually receive one RREP message in $2D$ hops. □

We continue with the proof that if two route discovery processes are executed concurrently, both of them will eventually terminate and at least a path for each is found.

Lemma 4.2.7 *For a $startSearch(j)_i$ input and a $startSearch(k)_l$ input, eventually corresponding $foundAnnouncement(j)_i$ and $foundAnnouncement(k)_l$ outputs occur.*

Proof. The proof of this lemma depends on the different possible paths built by these computations. We check the following cases:

- The two computations built baths with no node in common. That is $i \neq k \neq l$, $j \neq k \neq l$ and no intermediate node belongs in paths of both computations. In this case the paths built by the computations are distinct.

- The paths of the two computations have one intermediate node u in common, that is $u \neq i \neq j \neq k \neq l$. Then u can take distinct local decisions for each computation depending on their source and destination nodes. This is accomplished due to the distinct records in RT_u for each node. Thus the paths can be seen as distinct. The same holds in cases that the common node u is the source or destination node for the paths of the one computation and intermediate node for the paths of the other or if u is the source node of the paths of the one computation and the destination node for the paths of the other computation.

- The paths of the two computations have the same source or the same destination, that is $u = i = k$ or $u = j = l$. In these cases u can determine the paths for each computation by the pairs $(destId, destSeqNum)$ or $(srcId, srcSeqNum)$ respectively which are unique for each computation. Moreover if there is any intermediate node v, that is also in common, then v can determine the paths for the same source or destination by the different $(destId, destSeqNum)$ and $(srcId, srcSeqNum)$ pairs and the different records in RT_v for each source and destination (recall that for each destination, the relative record is unique by Lemma 1). Note that if v receives

two routes for the same source or same destination from different computations, then v holds the route with the larger $srcSeqNum$ or $destSeqNum$ respectively. Thus the paths can be built independently from each other.

Since in every case the two computations built their paths independently then we can use Lemma 4.2.6 for both computations and this completes the proof. □

Now, we are ready to conclude that every route discovery process executed in the system will eventually terminate, which is the proof that our specification solves the problem under study (i.e the routing problem).

Theorem 4.2.8 *For every $startSearch(j)_i$ input, eventually i performs a $foundAnnouncement(j)_i$ output.*

Proof. By Lemma 4.2.6 we get that when a single computation executes, then it terminates. By Lemma 4.2.7 we get that when any two computations execute, then the one does not involve in the process of the second, thus both eventually terminate. Since Lemma 4.2.7 holds for any pair of executions then it can be easily generalized to any number of executions. Thus from Lemmas 4.2.6 and 4.2.7 the result follows. □

Finally, we summarize our results to show that AOMDV exhibits the desired behavior.

Theorem 4.2.9 *If a route discovery process starts in a fair execution of algorithm AOMDV, then eventually it will terminate creating paths that are loop-free and link disjoint for the specific pair of source-destination nodes.*

Proof. From Theorems 4.2.4, 4.2.5 and 4.2.8. □

Chapter 5

Specification and Verification of AOMDV using Process Algebra

In this chapter we use the process algebra CCS for the specification and verification of AOMDV algorithm.

5.1 Specification of AOMDV

Data types:

$replies = \{rep \mid rep = \langle s, d, first, last\rangle\}$
$routes = \{r \mid r = \langle next, last, hops\rangle\}$ - List of different paths to a destination d
$RT = \{e \mid e = \langle d, dstSeq, Valid, hops, routes\rangle\}$ - The routing table of i
$B = \{t \mid t = \langle init, U, d, dstSeq, s, srcSeq, last, hops, j, e_s\rangle\}$
$C = \{t \mid t = \langle init, d, dstSeq, s, last, hops, j, e_d\rangle\}$
$RREQ_{j,i}(U, d, dstSeq, s, srcSeq, last, hops)$
$RREP_{j,i}(d, dstSeq, s, last, hops)$

$\text{System} \stackrel{\text{def}}{=} (\prod_{k\in K} \text{RP}[k, N_k, 0, \emptyset, \emptyset, \emptyset, \emptyset])\backslash L$
$L = \{RREQ_{i,j}, RREP_{i,j} | i \in K, j \in N_i\}$

$\text{RP}[i, N, seqNum, replies, RT, B, C] \stackrel{\text{def}}{=}$
$start_i(j).\mathsf{cnd}\ \big((\exists\, e \in RT \cdot (e{:}d = j \ \wedge\ e{:}Valid = \text{T})) \rhd$
$\text{RP}[\cdots, C \cup \langle \text{T}, j, -, i, -, -, -, -\rangle],$
$true \ \rhd\ \text{RP}[i, N, seqNum{+}1, \cdots,$
$B \cup \langle \text{T}, \neg(\exists\, e \in RT \cdot e{:}d = j), j, -, i, seqNum{+}1, -, -, -, -\rangle, C]\big)$
$+\ \sum_{j\in N} RREQ_{j,i}(U, d', dstSeq', s', srcSeq', last', hops').$
$\mathsf{cnd}\ \big((\exists\, e \in RT \cdot e{:}d = s') \rhd$

$$
\begin{array}{l}
\quad\quad \mathsf{cnd}\ ((e{:}dstSeq < srcSeq') \rhd \\
\quad\quad\quad\quad \mathrm{RP}[\cdots, RT \frown_e \langle s', srcSeq', \mathrm{T}, \infty, \langle j, f(i, last'), hops'{+}1\rangle\rangle, \\
\quad\quad\quad\quad\ B \cup \langle \mathrm{T}, U, d', dstSeq', s', srcSeq', f(i, last'), hops', j, e\ \rangle, C], \\
\quad\quad\quad\quad (e{:}dstSeq = srcSeq' \ \wedge\ e{:}hops > hops') \rhd \\
\quad\quad\quad\quad\ \mathsf{cnd}\ ((\nexists\, r \in e{:}routes \ \wedge\ (r{:}next = j \ \vee\ r{:}last = f(i, last'))) \rhd \\
\quad\quad\quad\quad\quad\quad \mathrm{RP}[\cdots, RT \cup\!\!\!\!\!\!\cdot_e \langle j, f(i, last'), hops'{+}1\rangle, \\
\quad\quad\quad\quad\quad\quad\ B \cup \langle \mathrm{F}, U, d', dstSeq', s', srcSeq', f(i, last'), hops', j, e\ \rangle, C], \\
\quad\quad\quad\quad\quad\quad true \ \rhd\ \mathsf{cnd}\ ((i = d') \rhd \\
\quad\quad\quad\quad\quad\quad\quad\quad \overline{RREP_{i,j}}(i, seqNum, s', \bot, 0).0 \mid \mathrm{RP}[\cdots, B, C] \\
\quad\quad\quad\quad true \ \rhd\ \mathrm{RP}[\cdots, B, C]) \\
\quad\quad true \ \rhd\ \mathrm{RP}[\cdots, B, C]) \\
true \rhd \mathrm{RP}[\cdots, RT \cup e = \langle s', srcSeq', \mathrm{T}, \infty, \langle j, f(i, last'), hops'{+}1\rangle\rangle, \\
\quad\quad B \cup \langle \mathrm{T}, U, d', dstSeq', s', srcSeq', f(i, last'), hops', j, e\ \rangle, C]\big) \\
+\ \textstyle\sum_{j \in N} RREP_{j,i}(d', dstSeq', s', last', hops'). \\
\mathsf{cnd}\ \big((\exists\, e \in RT \cdot e{:}d = d') \rhd \\
\quad\quad \mathsf{cnd}\ ((e{:}dstSeq < dstSeq') \rhd \\
\quad\quad\quad\quad \mathrm{RP}[\cdots, RT \frown_e \langle d', dstSeq', T, \infty, \langle j, f(i, last), hops'{+}1\rangle\rangle, B, \\
\quad\quad\quad\quad\ C \cup \langle \mathrm{T}, d', dstSeq', s', f(i, last), hops' + 1, j, e\rangle], \\
\quad\quad\quad\quad (e{:}dstSeq = dstSeq' \ \wedge\ e{:}hops > hops') \rhd \\
\quad\quad\quad\quad\ \mathsf{cnd}\ ((\nexists\, r \in e{:}routes \ \wedge\ (r{:}next = j \ \vee\ r{:}last = f(i, last'))) \rhd \\
\quad\quad\quad\quad\quad\quad \mathrm{RP}[\cdots, RT \cup\!\!\!\!\!\!\cdot_e \langle j, f(i, last), hops'{+}1\rangle, B, \\
\quad\quad\quad\quad\quad\quad\ C \cup \langle \mathrm{F}, d', dstSeq', s', f(i, last'), e{:}hops, j, e\rangle], \\
\quad\quad\quad\quad true \ \rhd\ \mathrm{RP}[\cdots, B, C] \\
\quad\quad\quad\quad true \ \rhd\ \mathrm{RP}[\cdots, B, C]) \\
\quad\quad true \rhd \mathrm{RP}[\cdots, RT \cup e = \langle d', dstSeq', T, \infty, \langle j, f(i, last), hops'{+}1\rangle\rangle, B, \\
\quad\quad C \cup \langle \mathrm{T}, d', dstSeq', s', f(i, last), hops' + 1, j, e\rangle]\big) \\
+ \textstyle\sum_{t \in B} \mathsf{cnd}\ \big((i = t{:}d \ \wedge\ t{:}init = \mathrm{T}) \rhd \\
\quad\quad\quad\quad \overline{RREP_{i,t:j}}(i, seqNum{+}1, t{:}s, \bot, 0).0 \\
\quad\quad\quad\quad\quad \mid \mathrm{RP}[i, N, seqNum{+}1, \cdots, B \backslash \{t\}, C], \\
\quad\quad\quad\quad (i = t{:}d \ \wedge\ t{:}init = \mathrm{F}) \rhd \\
\quad\quad\quad\quad \overline{RREP_{i,t:j}}(i, seqNum, t{:}s, \emptyset, 0).0 \mid \mathrm{RP}[\cdots, B \backslash \{t\}, C], \\
\quad\quad\quad\quad (\exists\, e \in RT \cdot (e{:}d = t{:}d \ \wedge\ e{:}Valid = \mathrm{T}) \wedge \\
\quad\quad\quad\quad \exists\, r \in e{:}routes \ \wedge\ \langle t{:}s, t{:}d, *, r{:}last\rangle \notin replies) \rhd \\
\quad\quad\quad\quad \overline{RREP_{i,t:j}}(t{:}d, e{:}dstSeq, t{:}s, r{:}last, maxHop(e)).0 \\
\quad\quad\quad\quad\quad \mid \mathrm{RP}[\cdots, replies \cup \langle t{:}s, t{:}d, t{:}last, r{:}last\rangle, RT \wr e, B \backslash \{t\}, C], \\
\quad\quad\quad\quad (i = t{:}s) \rhd \\
\quad\quad\quad\quad (\textstyle\prod_{k \in N} \overline{RREQ_{i,k}}(u(t{:}U, \exists\, e \in RT \cdot e{:}d = t{:}d), t{:}d, g(t{:}U, t, e), i,
\end{array}
$$

$$
\begin{array}{l}
\quad t{:}seqNum, \bot, 0).0 |\ \mathrm{RP}[\cdots, B \backslash \{t\}, C]), \\
\quad (t{:}init = T) \rhd \\
\quad (\prod_{k \in N} \overline{RREQ_{i,k}}(u(t{:}U, \exists\, e \in RT \cdot e{:}d = t{:}d), t{:}d, g(t{:}U, t, e), t{:}s, \\
\quad\ t{:}srcSeq, t{:}last, maxHop(t{:}e_s)).0 |\ \mathrm{RP}[\cdots, RT \wr t{:}e_s, B \backslash \{t\}, C]), \\
\quad true \ \rhd\ \mathrm{RP}[\cdots, B \backslash \{t\}, C]\big) \\
+ \sum_{t \in C} \mathsf{cnd}\ \big((i = t{:}s\ \wedge\ t{:}init = \mathrm{T})\ \rhd\ \overline{routeFound_i}(d).0 \mid \mathrm{RP}[\cdots, C \backslash \{t\}], \\
\quad (\exists\, e \in RT \cdot (e{:}d = t{:}s) \wedge \\
\quad\ \exists\, r \in e{:}routes\ \wedge\ \langle t{:}s, t{:}d, r{:}last, * \rangle \notin replies))) \rhd \\
\quad\ \overline{RREP_{i,r:next}}(t{:}d, t{:}dstSeq, t{:}s, t{:}last, maxHop(t{:}e_d)).0 \\
\quad\ \mid \mathrm{RP}[\cdots, replies \cup \langle t{:}s, t{:}d, r{:}last, t{:}last \rangle, RT \wr t{:}e_d, \cdots, C \backslash \{t\}], \\
\quad true \ \rhd\ \mathrm{RP}[\cdots, C \backslash \{t\}]\big)
\end{array}
$$

Functions:

$$
f(i,l) = \begin{cases} l & \text{if } l \neq \bot \\ i & \text{otherwise} \end{cases}
$$

$$
RT \frown_e \langle d, dstSeq, T, hops, \langle next, last, hops \rangle \rangle = \\
\{\cdots, e = \langle d, dstSeq, T, hops, \langle next, last, hops \rangle \rangle, \cdots\}
$$

$$
RT \cup_e \langle next, last, hops \rangle = \{\cdots, e{:}routes \cup \langle next, last, hops \rangle, \cdots\}
$$

$$
maxHop(e) = \begin{cases} e{:}hops & \text{if } e{:}hops \neq \infty \\ \max\{r{:}hops \mid r \in e{:}routes\} & \text{otherwise} \end{cases}
$$

$$
RT \wr e = \{\cdots, e{:}hops = maxHop(e), \cdots\}
$$

$$
u(U,l) = \begin{cases} \mathrm{T} & \text{if } U = \mathrm{T} \wedge l = \mathrm{F} \\ \mathrm{F} & \text{if } U = \mathrm{F} \vee l = \mathrm{T} \end{cases}
$$

$$
g(U,t,e) = \begin{cases} 0 & \text{if } u(U, \exists e \in RT \cdot e{:}d = t{:}d) = \mathrm{T} \\ t{:}dstSeq & \text{if } t{:}U = \mathrm{F} \wedge e \notin RT \\ e{:}dstSeq & \text{if } t{:}U = \mathrm{T} \wedge e \in RT \\ \max(t{:}dstSeq, e{:}dstSeq) & \text{if } t{:}U = \mathrm{F} \wedge e \in RT \end{cases}
$$

$$
\neg \ell = \begin{cases} \mathrm{T} & \text{if } \ell = \mathrm{F} \\ \mathrm{F} & \text{otherwise} \end{cases}
$$

The system model above consists of $|K|$ RP agents that correspond to the $|K|$ nodes of the network. Each node has its own id number i, its own sequence number $seqNum$, the set of its neighbors N and its routing table RT. Moreover, each process maintains three sets: the set *replies* that hold information about what replies were send for each path, the set B that holds the information from $RREQ$ messages that were received and i accepted their paths to the source and the set C that holds the information from $RREP$ messages that were received and i accepted their paths to the destination.

In brief, process RP$[i, \cdots]$ can do five actions:

1. Receive a signal $start_i(j)$ and start a routing discovery process.

2. Receive a message $RREQ$ and process it in order to decide if it will accept the path that contains the message to the source. If it accepts the path then it enters the information in a tuple in set B.

3. Receive a message $RREP$ and process it in order to decide if it will accept the path that contains the message to the destination. If it accepts the path then it enters the information in a tuple in set C.

4. Process any tuple in set B to decide its next actions including message forwarding and message reply.

5. Process any tuple in set C to decide its next actions including message forwarding and $\overline{routeFound_i}(j)$ action that terminates the route discovery process.

Here we give a more detailed explanation of our model in order to help the reader understand our notation and follow the correctness proof.

When a node i receives a signal $start_i(j)$ for a destination j it checks whether it already knows a valid path to j, that is $\exists\, e \in RT_i$ s.t. $e{:}d = j$ and $e{:}Valid = \mathrm{T}$. In this case it inserts a tuple t in set C with information about j in order to trigger the action $\overline{routeFound_i}(j)$ when t is processed. Otherwise, it increases its sequence number to denote a new path discovery process and inserts a tuple t in set B with the new sequence number and other prober information in order to trigger a message $RREQ$ to be sent to its neighbors when t is processed, starting in this way a path discovery process. If i knows a path to j but it is not valid then the tuple t contains the value $t{:}U = \mathrm{F}$ while if i does not know any path then $t{:}U = \mathrm{T}$.

Upon receiving a request message $RREQ$ for a route discovery process that was started by i for a destination j with source sequence number $srcSeq$, a node

k checks whether it knows any path to i, that is $\exists\, e : RT$ s.t. $e : d = i$. If yes then it reinitializes the record e if $srcSeq > e{:}dstSeq$ with the information $srcSeq$, $next$ (the node from which k received this message), $last$ and $hops + 1$. Moreover it sets the path as valid, $e : Valid = \mathrm{T}$, and the advertised hop count as infinity, $e : hops = \infty$. Since the path is accepted, a new tuple t is created with the information from the message and inserted in set B. Since this path resulted in reinitialization of the record for i (the message was the first one received for this route discovery process) then $t{:}init = \mathrm{T}$. Similar actions will be executed if k does not have any record for i, thus it does not know any path to i. In this case, a new record with the same information as above is created and inserted in the routing table RT. Moreover, the same tuple t will be inserted in B.

If k has a record $e \in RT$ s.t. $e{:}d = i$ and $srcSeq = e{:}dstSeq$, this means that the message received was not the first copy received for the specific route discovery process. In this case k accepts the new path only if the hops number in the message is smaller than the advertised hop count for i, $e{:}hops > hops$, a if there is no other route in $e{:}routes$ with the same $next$ and $last$ nodes. If the path is accepted, then a new route r is inserted in $e{:}routes$ with $r{:}hops = hops + 1$ and a new tuple t is inserted in B with the information of the message and $t{:}init = \mathrm{F}$ since the message did not result in initialization of the record e for i. If the path is not accepted then the message is regarded except if $k = d$. In this case a reply message is sent back to the node from which the message is received, but no route is inserted in $e{:}routes$ neither a tuple t is inserted in B.

Upon receiving a $RREP$ message, a node k checks whether it knows any path to the destination d and acts in a similar manner as in the case of a $RREQ$ message regarding the destination d instead of source s and the destination sequence number $dstSeq$ instead of the source sequence number $srcSeq$. Moreover, if the path is accepted, k inserts a tuple t in set C instead of set B.

If there exists a tuple $t \in B$, this means that t contains the information of a $RREQ$ message whose path was accepted. There are six cases:

- k is the destination of the message and the message is the first one received for the specific route discovery process, that is $k = t{:}d \wedge t{:}init = \mathrm{T}$. In this case, a reply message $RREP$ will be sent back to the node $t : j$ from which the message was received and the sequence number of k will be increased. The reply message includes this new destination sequence number. Finally t is removed from B.

- k is the destination but $t{:}init = \mathrm{F}$. In this case, a reply message $RREP$ will

be sent back to the node $t:j$ from which the message was received but the sequence number of k will not be increased. Finally t is removed from B.

- If k is not the destination node but it knows a valid path to the destination, $\exists\, e \in RT$ s.t. $e:d = d$ and $e:Valid = \mathrm{T}$, then it checks if there is any route to the destination that was not included in any previous reply to this source, that is $\exists\, r \in e:routes$ s.t. $\langle t:s, t:d, *, t:last\rangle \notin replies$. In this case k sends a reply message to $t:j$ with route r and the information it has for d, like the destination sequence number $e:dstSeq$ and the advertised hop count. If $e:hops = \infty$ then it sets the advertised hop count to the maximum hop count of the routes to d. Finally it enters the appropriate information in $replies$ set and t is removed from B.

- If k is the source, $k = t:s$ then it sends a $RREQ$ message to all its neighbors. The message contains the value $U = \mathrm{T}$ if there is no record $e \in RT$ for the destination and $U = \mathrm{F}$ if there is a record. In this case $dstSeq = e:dstSeq$, otherwise $dstSeq = 0$.

- If none of the above holds and $t:init = \mathrm{T}$, this denotes that k is an intermediate node that does not know a valid path to the destination $t:d$. Thus, k sends a $RREQ$ message to all its neighbors including the information in t. The message contains the value $U = \mathrm{T}$ if $t:U = \mathrm{T}$ and there is no record $e \in RT$ for the destination. Otherwise $U = \mathrm{F}$ and $dstSeq = \max\{e:dstSeq, t:dstSeq\}$ if $\exists\, e \in RT$. Moreover, if $e:hops = \infty$ then it sets the advertised hop count to the maximum hop count of the routes to $t:s$. Finally it exits t from B.

- If none of the above holds then t is removed from B.

If there exists a tuple $t \in C$, this means that t contains the information of a $RREP$ message whose path was accepted. There are three cases:

- If k is the source of the route discovery process and $t:init = \mathrm{T}$. This means that t is the first tuple created for this purpose. In this case k will send a signal $\overline{routeFound_k}(t:d)$ and remove t from C.

- If k is not the destination node but it knows a path to the source, $\exists\, e \in RT$ s.t. $e:d = s$ and a route r that was not used in any previous reply from d to s, that is $\exists\, r \in e:routes$ s.t. $\langle t:s, t:d, t:last, *\rangle \notin replies$, then k sends a reply message to $r:next$ with the information it has for d, like the destination sequence number $e:dstSeq$ and the advertised hop count. Again,

if $e:hops = \infty$, then it sets the advertised hop count to the maximum hop count of the routes to d. Finally it enters the appropriate information in *replies* set and exits t from C.

- If none of the above holds then t is removed from C.

With this action, the explanation of our model is completed.

5.2 Correctness Proof

In this section we aim to prove that *System* exhibits the desired behavior. To achieve this we will prove that *System* is weakly bisimilar to its specification which shows the desired behavior.

To begin with, let P a derivative of *System* such that

$$\begin{aligned} P = \quad & \big(\mathrm{RP}[1, N_1, seqNum_1, replies_1, RT_1, B_1, C_1] \\ & \mid \cdots \\ & \mid \mathrm{RP}[n, N_n, seqNum_n, replies_n, RT_n, B_n, C_n]\big)\backslash L \end{aligned}$$

This notation will be used hereafter.

Let us also define a useful function $Paths(P, i, x)$ that returns the set of paths from a node i to a node x, $i, x \in K$, at a given state P by extracting the paths from the routing tables of the nodes in the paths recursively as follows

$$Paths(P,i,x) = \begin{cases} \{\langle i\rangle +\!\!+ Paths(P,j,x) \mid & \exists\, e \in RT_i \cdot (e{:}d = x) \wedge \\ & r \in e{:}routes \ \wedge\ r{:}next = j\}, i \neq x \\ \{\langle x\rangle\}, & i = x \end{cases}$$

Consider the following process

$$\begin{aligned} Spec_S = & \textstyle\sum_{i,d} start_i(d).Spec_{S\cup\{\langle i,d\rangle\}} \\ & + \textstyle\sum_{\{\langle i,d\rangle\}\in S} \overline{routeFound_i}(d).Spec_{S-\{\langle i,d\rangle\}} \end{aligned}$$

To prove that *System* modeled in Section 5.1 delivers the desired behavior, we have to show that:

Theorem 5.2.1

1. $System = Spec_\emptyset$
2. *For any* P *such that* $System \Longrightarrow P$ *and* $\forall\ i, x \in K,\ Paths(P, i, x)$ *is a set of loop-free and link disjoint paths.*

We start with the proof of the second part of Theorem 5.2.1, that is, at any state of the system, the paths between any pair of nodes are loop free and link disjoint.

We start with three lemmas that we will use later in our proof. Their proof is easily derived from the specification of the system.

The first lemma says that when a communication happens regarding a path to a destination, then the sender process sets the hops for this destination to be equal to the value on the message.

Lemma 5.2.2 *If* $P \xrightarrow{\tau} P'$ *such that*

- τ *has arisen from a communication along a channel* $RREQ_{i,j}, i \neq s$ *with values* $s, srcSeq$ *and hops or*
- τ *has arisen from a communication along a channel* $RREP_{i,j}, i \neq d$ *with values* $d, dstSeq$ *and hops*

then $\exists e \in RT_i \cdot (e{:}d{=}s \wedge e{:}dstSeq = srcSeq)$ *or* $\exists e \in RT_i \cdot (e{:}d = d \wedge e{:}dstSeq = dstSeq)$ *respectively, and* $e{:}hops = hops$.

The second lemma states that a path increases its hops number as new nodes are inserted in the path.

Lemma 5.2.3 *If* $P \xrightarrow{\tau} P'$ *such that*

- τ *has arisen from a communication along a channel* $RREQ_{i,j}$ *with values* $d, s, srcSeq, last$ *and hops or*
- τ *has arisen from a communication along a channel* $RREP_{i,j}$ *with values* $d, dstSeq, s, last$ *and hops*

and $P' \Rightarrow\!\!\xrightarrow{\tau} P''$ *such that*

- τ *has arisen from a communication along a channel* $RREQ_{x,y}$ *with values* $d, s, srcSeq, last$ *and* $hops'$ *or*
- τ *has arisen from a communication along a channel* $RREP_{x,y}$ *with values* $d, dstSeq, s, last$ *and* $hops'$

then $hops' > hops$.

The third lemma shows that when the hops number for a destination on a node i is set to a numerical value for a specific destination sequence number $dstSeq$, it does not change as long as $dstSeq$ remains the same.

Lemma 5.2.4 *If* $RP[i, \cdots, RT, \cdots]$ *and* $\exists\, e = \langle d, dstSeq, \cdots, hops, \cdots\rangle \in RT$, *such that* $hops \neq \infty$ *then if* $RP[i, \cdots, RT, \cdots] \xrightarrow{\alpha} RP[i, \cdots, RT', \cdots]$ *where*

- $\alpha \neq RREQ_{j,i}(\cdots, s, srcSeq, \cdots)$, $s = e{:}d$, $srcSeq > e{:}dstSeq$ *or*
- $\alpha \neq RREP_{j,i}(\cdots, d, dstSeq, \cdots)$, $d = e{:}d$, $dstSeq > e{:}dstSeq$

then for $e' = \langle d, dstSeq, \cdots, hops, \cdots\rangle \in RT'$, $e{:}hops = e'{:}hops$.

We, now, continue with our main lemmas.

Lemma 5.2.5 *Suppose* $System \stackrel{\omega}{\Longrightarrow} P$ *and let* $i, j \in K$. *Then for any* $p \in Paths(P, i, j)$, p *is a loop free path.*

Proof. Let a path $p = i \ldots k \underbrace{k_1, \cdots, k_n} k \ldots j \in Paths(P, i, j)$. This means that there exists an execution $\omega = \alpha_{k,k_1} \ldots \alpha_{k_{n-1},k_n}$ such that $System \Longrightarrow Q \xrightarrow{\omega} Q' \xrightarrow{\alpha_{k_n,k}} P$ where at state Q, node k enters in the path for the first time and P is the state where the path creates a cycle. Here we check the case that each $\alpha_{u,v}$ is a communication along a channel $RREQ_{u,v}$ with values $d, s, srcSeq, last$ and $hops$. The same arguments can be applied in the case that $\alpha_{u,v}$ is a communication along a channel $RREP_{k,k_1}$ with values $d, dstSeq, s, last$ and $hops$.

α_{k,k_1} is a communication along the channel $RREQ_{k,k_1}$ with values d, s, $srcSeq$, $last$ and $hops$. According to Lemma 5.2.2, for $e \in RT_k$ s.t. $e : d = s$ and $e{:}dstSeq = srcSeq$, it holds that $e{:}hops = hops$.

$\vdots$

$\alpha_{k_n,k}$ is a communication along a channel $RREQ_{k_n,k}$ with values d, s, $srcSeq$, $last$ and $hops'$. According to Lemma 5.2.3, it holds that $hops' > hops$. Moreover, according to Lemma 5.2.4, for $RP[k, \cdots, RT, \cdots]$, and $e' = \langle s, srcSeq, \cdots, hops, \cdots\rangle \in RT$, it holds that $e' : hops = e : hops$, thus $e'{:}hops < hops'$. Now, from the specification of the system, we see that this path received by a message $RREQ$ such that $srcSeq = e'{:}dstSeq$ is not accepted by k since $hops' > e'{:}hop$. Thus, $\langle k_n, last, hops'{+}1\rangle \notin e'{:}routes$ and $Q' \xrightarrow{\alpha_{k_n,k}} Q'' \neq P$. This leads to the conclusion that $Q' \xrightarrow{\alpha_{k_n,k}} P$ is impossible.

□

Lemma 5.2.6 *For every* P, i, x *and for each* $p_1, p_2 \in Paths(P, i, x)$, p_1, p_2 *are link disjoint.*

Proof. The proof is by induction on the length of the transition $System \Longrightarrow P$. For

$$P_0 = \big(\prod_{k \in K} RP[k, N_k, 0, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset] \big) \backslash L$$

It is easy to see that $\forall i, x \; Paths(P_0, i, x) = \emptyset$, thus the proof is trivial.

Let the proposition hold for a state P' such that $P_0 \Rightarrow P'$. We will prove that the proposition holds for a state P'' such that $P' \xrightarrow{\alpha} P''$. The proof is by case analysis on action α.

- If $\alpha = start(d)$ we can see that $\forall i, x \; Paths(P'', i, x) = Paths(P', i, x)$ thus for any $p_1, p_2 \in Paths(P', i, x)$, p_1, p_2 are link disjoint by the induction hypothesis.
- The same holds for $\alpha = \overline{routeFound}(d)$.
- If $\alpha = \tau$ and the action has arisen from a communication along a channel $RREQ_{j,i}$ then there exist the following cases:
  - First note that $\forall x, \; Paths(P'', j, x) = Paths(P', j, x)$ thus for any $p_1, p_2 \in Paths(P'', j, x)$, p_1, p_2 are link disjoint.
  - If in state P' it holds that $\nexists \; e \in RT_i$ such that $e : d = j$ then $Paths(P'', i, j) = \langle j, i \rangle$ which is link disjoint by trivial.
  - If in state P' it holds that $\exists \; e \in RT_i$ such that $e : d = s'$ then $Paths(P'', i, j) = Paths(P', i, j)$ which is link disjoint from our induction hypothesis.
  - If in state P' it holds $\nexists \, e \in RT_i$ such that $e : d = s'$ then $Paths(P'', i, s') = \langle j, f(i, last') \rangle$ and $\forall x \neq s', j$, $Paths(P'', i, x) = Paths(P', i, x)$. Thus for any $x, p_1, p_2 \in Paths(P'', i, x)$, p_1, p_2 are link disjoint.
  - The same as above holds in case that in $P' \; \exists \; e \in RT_i$ s.t. $e : d = s'$ and $e : dstSeq < srcSeq'$.
  - If in state P' holds that $\exists e \in RT_i$ s.t. $e : d = s' \; \wedge \; e : dstSeq = srcSeq'$ and $e : hops > hops'$ and $\exists \; r \in e : routes$ s.t. $(r : next = j \; \vee \; r : last = f(i, last'))$ then $Paths(P'', i, s') = Paths(P', i, s')$ and $\forall x \neq s', j$, $Paths(P'', i, x) = Paths(P', i, x)$. Thus for any $x, p_1, p_2 \in Paths(P'', i, x)$, p_1, p_2 are link disjoint.
  - The last case is the case where in state P' it holds that $\exists e \in RT_i$ s.t. $e : d = s'$ and $e : dstSeq = srcSeq' \; \wedge \; e : hops > hops'$ but $\nexists \; r \in e : routes$ s.t. $(r : next = j \; \vee \; r : last = f(i, last'))$ then

$Paths(P'', i, s') = Paths(P', i, s') \cup \langle j, f(i, last')\rangle$ and $\forall x \neq s', j$, $Paths(P'', i, x) = Paths(P', i, x)$. We prove that for any $x, p_1, p_2 \in Paths(P'', i, s')$, p_1, p_2 are link disjoint by contradiction. Assume that $Paths(P'', i, s')$ has two paths p, p' that are not link disjoint. This means there is a link $(u, v), u, v \neq j, f(i, last)$ common in p, p'. Since $u, v \neq j, f(i, last)$ then $p, p' \neq \langle j, f(i, last')\rangle$ (the new inserted path). Thus $p, p' \in Paths(P', i, s')$ which is a contradiction to our induction hypothesis. Thus for any $x, p_1, p_2 \in Paths(P'', i, x)$, p_1, p_2 are link disjoint and the proof for this action is complete.

- If $\alpha = \tau$ and the action has arisen from a communication along a channel $RREP_{j,i}$ then the proof follows similarly to the previous case.

□

We continue with the proof of the first part of Theorem 5.2.1. In this proof we consider a restricted system similar to $System$ which delivers the desired behavior. Then we show that $System$ simulates the restricted system, thus it can also deliver the desired behavior. Furthermore we show that every derivative of $System$ can deliver the desired behavior. To conclude we show that $System$ is weak bisimilar to the process $Spec_{\emptyset}$, in other words, $System$ always delivers the desired behavior.

Consider a restricted system as follows:

Let $\{w_1, \cdots, w_n\}$ a set of loop-free and link-disjoint paths. Let $Flow(s, d) = \{w = \langle s, \cdots, d\rangle \mid w \in \{w_1, \cdots, w_n\}\}$ and $MainPaths(s, d) \subseteq Flow(s, d)$ such that for every pair of paths $w_1, w_2 \in MainPaths(s, d)$, if $i \neq s \wedge i \neq d \ \wedge \ i \in w_1 \wedge i \in w_2$ then either $w_1 = w_2$. Moreover, for each node i that participates in a path $w \in MainPaths(s, d)$ name w as w^i_{main}. If a node does not participate in any path $w \in MainPaths(s, d)$, but participates in some $w \in Flow(s, d)$ then $w^i_{main} = w$.

Finally, let

$$main_i = \{\langle s, prev, next, d\rangle \mid w = \langle s, \ldots, prev, i, next \ldots, d\rangle \in Flow(s, d), \\ \forall s, d \in K, \wedge w = w^i_{main}\}$$

$$accepted = \{\langle s, first, prev, next, last, d\rangle \mid \\ \langle s, first, \ldots, prev, i, next \ldots, last, d\rangle \in Flow(s, d), \ \forall s, d \in K\}$$

$$N_{next}(i, s, d) = \{j \mid \exists \langle s, \cdots, next = j, \cdots, d\rangle \in accepted\}$$

The set $accepted_i$ keeps the loop-free and link-disjoint paths that i participates in. From this set we extract the neighbors N_{next} that should be informed if a RREQ message is received for the specific source-destination pair (s, d). Finally set $main_i$ contains a copy of each path in $accepted_i$ that is considered as main path for a source-destination pair.

Next we give the form of the agents we use in our restricted system. Wherever the new agent acts as agent RP we put $\cdots$ while the differences between them are shown with bold fonts.

$$
\begin{array}{l}
\mathrm{RP}'[i, N, main, accepted, seqNum, replies, RT, B, C] \stackrel{\text{def}}{=} \\
\quad start_i(j) \cdots \\
\quad + \sum_{j \in N} RREQ_{j,i}(U, d, dstSeq, s, srcSeq, last, hops). \\
\quad\ \mathsf{cnd}\ \big((\exists\, e \in RT \cdot e{:}d = s) \rhd \\
\qquad \mathsf{cnd}\ ((e{:}dstSeq < srcSeq)\ \rhd\ \cdots \\
\qquad\quad (\mathbf{e{:}dstSeq = srcSeq}) \rhd \\
\qquad\quad\ \mathrm{RP}'[\cdots, RT \underset{e}{\cup} \langle j, f(i, last), hops{+}1\rangle, \\
\qquad\quad\quad B \cup \langle \mathrm{F}, U, d, dstSeq, s, srcSeq, f(i, last), hops, j, e\rangle, C], \\
\qquad\quad true\ \rhd\ \mathrm{RP}'[\cdots, B, C]) \\
\qquad true\ \rhd\ \cdots\big) \\
\quad + \sum_{j \in N} RREP_{j,i}(d, dstSeq, s, last, hops). \\
\quad\ \mathsf{cnd}\ \big((\exists\, e \in RT \cdot e{:}d = d) \rhd \\
\qquad \mathsf{cnd}\ ((e{:}dstSeq < dstSeq)\ \rhd\ \cdots \\
\qquad\quad (\mathbf{e{:}dstSeq = dstSeq}) \rhd \\
\qquad\quad\ \mathrm{RP}'[\cdots, RT \underset{e}{\cup} \langle j, f(i, last), hops{+}1\rangle, B, \\
\qquad\quad\quad C \cup \langle \mathrm{F}, d, dstSeq, s, f(i, last), e{:}hops, j, e\rangle], \\
\qquad\quad true\ \rhd\ \mathrm{RP}'[\cdots, B, C]) \\
\qquad true\ \rhd\ \cdots\big) \\
\quad + \sum_{t \in B} \mathsf{cnd}\ \big((i = t{:}d\ \wedge\ \langle \mathbf{t{:}s}, \mathbf{t{:}j}, -, \mathbf{t{:}d}\rangle \in \mathbf{main}) \rhd \\
\qquad\quad \overline{RREP_{i,t:j}}(\cdots).0 |\ \mathrm{RP}'[\cdots, seqNum{+}1, \cdots, B \backslash \{t\}, C], \\
\qquad\quad (i = t{:}d\ \wedge\ \langle \mathbf{t{:}s}, *, \mathbf{j}, -, \mathbf{j}, \mathbf{d}\rangle \in \mathbf{accepted}) \rhd \\
\qquad\quad\ \overline{RREP_{i,t:j}}(\cdots).0\ |\ \mathrm{RP}'[\cdots, B \backslash \{t\}, C], \\
\qquad\quad (\exists\, e \in RT \cdot (e{:}d = t{:}d\ \wedge\ e{:}Valid = \mathrm{T}) \wedge \\
\qquad\quad\ \exists\, r \in e{:}routes\ \wedge\ \langle \mathbf{t{:}s}, *, \mathbf{t{:}j}, \mathbf{r{:}next}, \mathbf{r{:}last}, \mathbf{t{:}d}\rangle \in \mathbf{accepted}) \rhd \\
\qquad\quad\quad \overline{RREP_{i,t:j}}(\cdots).0 |\ \mathrm{RP}'[\cdots, RT \wr e, B \backslash \{t\}, C], \\
\qquad\quad (i = t{:}s)\ \rhd\ (\prod_{\mathbf{k} \in \mathbf{N_{next}(i)}} \overline{RREQ_{i,k}}(\cdots).0 |\ \mathrm{RP}'[\cdots, B \backslash \{t\}, C]), \\
\qquad\quad \langle \mathbf{t{:}s}, \mathbf{t{:}j}, *, \mathbf{t{:}d}\rangle \in \mathbf{main} \rhd \\
\qquad\quad\ (\prod_{\mathbf{k} \in \mathbf{N_{next}(i)}} \overline{RREQ_{i,k}}(\cdots).0 |\ \mathrm{RP}'[\cdots, RT \wr t{:}e_s, B \backslash \{t\}, C]), \\
\qquad\quad true\ \rhd\ \mathrm{RP}'[\cdots, B \backslash \{t\}, C]\big) \\
\quad + \sum_{t \in C} \mathsf{cnd}\ \big((i = t{:}s\ \wedge\ \langle \mathbf{t{:}s}, *, \mathbf{t{:}j}, \mathbf{t{:}d}\rangle \in \mathbf{main}) \rhd \\
\qquad\quad \overline{routeFound_i}(d).0\ |\ \mathrm{RP}'[\cdots, C \backslash \{t\}], \\
\qquad\quad (\exists\, e \in RT \cdot (e{:}d = t{:}s) \wedge \\
\qquad\quad\ \exists\, r \in e{:}routes \cdot \langle \mathbf{t{:}s}, *, \mathbf{r{:}next}, \mathbf{t{:}j}, \mathbf{t{:}last}, \mathbf{t{:}d}\rangle \in \mathbf{accepted}) \rhd \\
\qquad\quad\quad \overline{RREP_{i,r:next}}(\cdots).0 |\ \mathrm{RP}'[\cdots, RT \wr t{:}e_d, \cdots, C \backslash \{t\}],
\end{array}
$$

$$true \ \triangleright \ \mathrm{RP}'[\cdots, C\backslash\{t\}])$$

Thus RP′ is similar to RP except that it restricts its outputs over a set of loop free and link disjoint paths.

Let $\mathcal{T}$ be the set of agents of the form

$T_0 \stackrel{\text{def}}{=} (\prod_{k\in K} \mathrm{RP}'[k, N, main, accepted, seqNum, replies, RT, B, C])\backslash L$

We prove that the restricted systems T_0 can deliver the desired behavior, that is, every request for a path to some destination is eventually satisfied.

Lemma 5.2.7 $T_0 \stackrel{start_s(d)}{\longrightarrow} T_1 \stackrel{\overline{routeFound_s}(d)}{\Longrightarrow} T_w$.

Proof. The proof is by construction of the possible execution of T_0 that leads to the desired result. Let us start with $T_0 \stackrel{start_s(d)}{\longrightarrow} T_1$ and let $D = \{u \in K \mid u = d \vee (\exists e \in RT_u \wedge e{:}d = d \wedge e{:}Valid = \mathrm{T})\}$. If $s \in D$ then $T_1 \stackrel{\overline{routeFound_s}(d)}{\longrightarrow} T_2$ as asked. We continue for the case that $s \notin D$.

Let $p \in MainPaths(s, d)$ the path with the minimum distance L from s to node $u \in D$. Name the nodes $v_0, v_1, \ldots, v_L$ such that each node $v_l, 0 \le l \le L$ is at distance l from s on path p. Note that $v_0 = s$, $v_L = u$ and for each v_l,

$\langle v_0, v_{l-1}, v_{l+1}, d\rangle \in main_{v_l}, \quad 0 \le l \le L$ and
$\langle v_0, v_1, v_{l-1}, v_{l+1}, p{:}last, d\rangle \in accepted_{v_l}, 0 \le l \le L$
where $v_{l-1} = \perp$ if $l = 0$ and
$v_{l+1} = \perp$ if $v_l = d$

Now, let us write T^l, $0 \le l \le L$, for the process

$$\begin{aligned} T^l \stackrel{\text{def}}{=} (&\textstyle\prod_{i\notin\langle v_0,\ldots,v_L\rangle} \mathrm{RP}'[i, N, main, accepted, seqNum, replies, RT, B, C] \\ &\mid \textstyle\prod_{i\in\{v_0,\ldots,v_L\}-v_l} \mathrm{RP}'[i, N, main, accepted, seqNum, replies, RT, B, C] \\ &\mid \mathrm{RP}'[v_l, N, main, accepted, seqNum, replies, RT, \\ &\ B \cup \langle init, d, dstSeq, s, srcSeq, v_1, hops, v_{l-1}, e\rangle, C])\backslash L \end{aligned}$$

That is, in step l, the node in distance l receives the request and is ready for its next action. Note that $T^0 = T_1$. It is easy to see that $T^l \Longrightarrow T^{l+1}, 0 \le l \le L-1$ since for each v_l, $\langle v_0, v_{l-1}, v_{l+1}, d\rangle \in main_{v_l}$ for each v_{l+1}, $\langle v_0, *, v_l, v_{l+2}, p{:}last, d\rangle \in accepted_{v_{l+1}}$ and if $\exists\, e \in RT_{v_{l+1}}$ s.t. $e{:}d = s$ then $e{:}dstSeq < srcSeq$, $0 \le l \le L-1$. Thus,

$$\mathrm{RP}'[v_l, \cdots, B\cup\langle init, d, dstSeq, s, srcSeq, v_1, hops, v_{l-1}, e\rangle, C] \stackrel{\overline{RREQ_{v_l,v_{l+1}}}(U,d,destSeq,s,srcSeq,v_1,hops)}{\longrightarrow} \mathrm{RP}'[v_l, \cdots, B, C]$$

$$\mathrm{RP}'[v_{l+1}, \cdots, B, C] \stackrel{RREQ_{v_l,v_{l+1}}(U,d,destSeq,s,srcSeq,v_1,hops)}{\longrightarrow} \mathrm{RP}'[v_{l+1}, \cdots, B\cup\langle init, d, dstSeq, s, srcSeq, v_1, hops, v_l, e\rangle, C]$$

When $l = L$ the request has arrived at a node $v_L = u \in D$, that means, a reply message will be generated since $\langle s, *, v_{L-1}, v_{L+1}, p{:}last, d\rangle \in accepted_{v_L}$. Thus,

$$\mathrm{RP}'[v_L, \cdots, B \cup \langle init, d, dstSeq, s, srcSeq, v_1, hops, v_{L-1}, e\rangle, C] \xrightarrow{\overline{RREP_{v_L, v_{L-1}}}(d, dstSeq', s, p:last, hops)} \mathrm{RP}'[v_L, \cdots, B, C]$$

$$\mathrm{RP}'[v_{L-1}, \cdots, B, C] \xrightarrow{RREP_{v_L, v_{L-1}}(d, dstSeq', s, p:last, hops)} \mathrm{RP}'[v_{L-1}, \cdots, B, C \cup \langle init, d, dstSeq', s, p{:}last, hops, v_L, e\rangle]$$

since $\langle s, *, v_{l-2}, v_l, p{:}last, d\rangle \in accepted_{v_{l-1}}$, $1 \leq l \leq L$.

Let us write R^l, $0 \leq l \leq L-1$ for the process

$$\begin{aligned} R^l \stackrel{\text{def}}{=} (&\textstyle\prod_{i \notin \langle v_0, \ldots, v_L\rangle} \mathrm{RP}'[i, N, main, accepted, seqNum, replies, RT, B, C] \\ &\mid \textstyle\prod_{i \in \{v_0, \ldots, v_{L-1}\} - v_l} \mathrm{RP}'[i, N, main, accepted, seqNum, replies, RT, B, C] \\ &\mid \mathrm{RP}'[v_l, N, main, accepted, seqNum, replies, RT, B, \\ &\quad C \cup \langle init, d, dstSeq, s, p{:}last, hops, v_{l+1}, e\rangle])\backslash L \end{aligned}$$

That is, in step l, the node in distance l from s, receives the reply message. We have seen how $T^L \Longrightarrow R^{L-1}$. With similar arguments we prove that for any l, $2 \leq l \leq L-1$, $R^l \Longrightarrow R^{l-1}$ since for each v_l, $\langle s, *, v_{l-1}, v_{l+1}, p{:}last, d\rangle \in accepted_{v_l}$ and for each v_{l-1}, $\langle s, *, v_{l-2}, v_l, p{:}last, d\rangle \in accepted_{v_{l-1}}$, $1 \leq l \leq L-1$.

When we reach $l = 0$, $v_0 = s$ will receive the reply message:

$$\mathrm{RP}'[s, \cdots, C \cup \langle init, d, dstSeq, s, last, hops, v_1, e\rangle] \xrightarrow{\overline{routeFound_s}(d)} \mathrm{RP}'[s, \cdots, C]$$

that is

$$T_1 \xrightarrow{\overline{routeFound_s}(d)} T_x$$

After this point we can apply similar arguments for all paths $p' \in MainPaths(s, d)$ except from the last step above. In these cases for $l = 0$ it would be:

$$\mathrm{RP}'[s, \cdots, C \cup \langle init, d, dstSeq, s, last, hops, v_1, e\rangle] \longrightarrow \mathrm{RP}'[s, \cdots, C]$$

When finished for every path $p' \in MainPaths(s, d)$, it holds that

$$T_x \Longrightarrow T_y$$

where $Paths(T_y, s, d) = MainPaths(s, d)$. In case that $MainPaths(s, d) = Flow(s, d)$ then $Paths(T_y, s, d) = Flow(s, d)$ as asked.

In case that $\exists\, p \in Flow(s,d) \setminus MainPaths(s,d)$ then for any such path, there is a similar $RREQ$ sequence of actions as the previous paths, until the message reaches a node v that also participates in a path that is already built. That is, $\exists\, e \in RT_v \wedge e{:}d = s \wedge e{:}dstSeq = srcSeq$. In this case the message is not forwarded but the path is inserted in RT_v. If a $RREP$ message on p was previously received by v or when such message is received, v creates a $RREP$ message which sends backwards in p until it reaches s.

When this is done for every path p, then

$$T_y \Longrightarrow T_w$$

where $Paths(T_w, s, d) = Flow(s, d)$ and the proof is completed. □

We now give an important feature of our restricted system which will help us in the sequel of the proof.

Lemma 5.2.8 *T_0 is confluent.*

Proof. The process RP′ is confluent by construction and input-enabled. Moreover, each channel employed in T_0 is used by at most two processes in the system. Thus, by Theorem 2.2.6, the result follows. □

For our next result we need a definition of the pending requests of a computation.

Definition 5.2.9 *For any derivative T of T_0 such that $T_0 \stackrel{\omega}{\Longrightarrow} T$,*

$$pending(T) = \{\langle s,d\rangle \mid start_s(d) \in \omega \text{ and } \overline{routeFound_s}(d) \notin \omega, \forall\, s,d \in K\}$$

Similarly we define $pending(P)$ for any derivative P of $System$.

Now, we can prove that our restricted system delivers the required behavior.

Lemma 5.2.10 *Suppose T is a derivative of T_0 and suppose $\langle i,d\rangle \in pending(T)$. Then $T \stackrel{\overline{routeFound_i}(d)}{\Longrightarrow} T'$.*

Proof. Suppose $T_0 \stackrel{\omega}{\Longrightarrow} T$. Then it must be that $start_i(d) \in \omega$. Specifically, there exists some T_1 such that $T_0 \stackrel{\omega_1}{\Longrightarrow} T_1 \stackrel{start_i(d)}{\longrightarrow} T_2 \stackrel{\omega_2}{\Longrightarrow} T$. By Lemma 5.2.7 $T_2 \stackrel{\overline{routeFound_i}(d)}{\Longrightarrow} T_3$. Furthermore, since T_0 is confluent, there exists T_3', T' such that $T_3 \stackrel{\omega_2}{\Longrightarrow} T_3'$, $T \stackrel{\overline{routeFound_i}(d)}{\Longrightarrow} T'$ and $T_3' \approx T'$. Thus, the result follows. □

So far, we have created a restricted system and we have proved that every request for a path to some destination is eventually satisfied. It now remains to

relate *System* to T_0 in order to apply the above results. First we built a notion of similarity between derivatives of T_0 and *System* which we will use in the relation between *System* and T_0.

Definition 5.2.11 *Let*

$$T \stackrel{\text{def}}{=} (\prod_{i \in K} \mathrm{RP}'[i, main_i, accepted_i, seqNum_i, replies_i, RT'_i, B'_i, C'_i])$$

$$P \stackrel{\text{def}}{=} (\prod_{i \in K} \mathrm{RP}[i, N_i, seqNum_i, replies_i, RT_i, B_i, C_i])$$

where

$B'_i \subseteq B_i,\ C'_i \subseteq C_i,$
if $\exists\, e \in RT'_i$ *then* $e \in RT_i$, *and for any* $r \in e.routes$ *s.t.* $e \in RT'_i$, *then* $r \in e.routes$ *s.t.* $e \in RT_i$,
$N_i \supseteq \{u, u' \in K \mid \langle *, *, u, u', *, * \rangle \in accepted_i\}$ *and*
$Paths(T, s, d) = Paths(P, s, d)$.

Then we say that P *and* T *are similar processes.*

Now we prove that there exists a simulation relation between T_0 and *System*, specifically *System* simulates T_0. In other words, *System* can do anything T_0 can do, while *System* can do more things.

Lemma 5.2.12 $\mathcal{R} = \{\langle T, P\rangle \mid P$ *and* T *are similar*$\}$ *is a strong simulation.*

Proof. Let $\langle T, P\rangle \in \mathcal{R}$ and $T \xrightarrow{a} T'$. We will show that $P \xrightarrow{a} P'$ and $\langle T', P'\rangle \in \mathcal{R}$. The proof is a case analysis on the possible actions of T.

- If $a = start_i(j)$, $j \in K - \{i\}$, there are two possible cases:

  1. $\exists\, e \in RT_i \ \wedge\ e{:}d = j \ \wedge\ e{:}Valid = \mathrm{T}$

     $$\mathrm{RP}'[i, \cdots] \xrightarrow{start_i(j)} \mathrm{RP}'[i, \cdots, C \cup \langle \mathrm{T}, j, -, i, -, -, -, -\rangle]$$

  2. $\nexists\, e \in RT_i$ *s.t.* $e{:}d = j$ or $\exists\, e \in RT_i \ \wedge\ e{:}d = j \ \wedge\ e{:}Valid = \mathrm{F}$

$$\mathrm{RP}'[i, \cdots] \xrightarrow{start_i(j)} \mathrm{RP}'[i, \cdots, B \cup \langle \mathrm{T}, (e \in RT), j, -, i, seqNum_i{+}1, -, -, -, -\rangle, C]$$

  In both cases $P \xrightarrow{a} P'$ and $\langle T', P'\rangle \in \mathcal{R}$.

- If $a = \overline{routeFound_i}(j)$, $j \in K - \{i\}$ then:

$$\mathrm{RP}'[i, \cdots, C \cup \langle \mathrm{T}, j, *, i, *, *, *, * \rangle] \xrightarrow{\overline{routeFound_i}(j)} \mathrm{RP}'[i, \cdots, C]$$

which can be followed by process RP_i of P since C_i set of RP'_i is subset of the corresponding C_i set of RP_i. Thus $P \xrightarrow{a} P'$ and $\langle T', P' \rangle \in \mathcal{R}$.

- If $a = \tau$ and the action has arisen from a communication along a channel of type $RREQ$ between processes RP'_i and RP'_j, $i, j \in K$. For process RP'_i there are two possible cases:

  1. $i = s$. This means that i has previously received a *start* signal for a destination d and it does not have any valid route to this destination. Moreover, it holds that $\langle i, j, -, j, *, d \rangle \in accepted_i$ and $j \in N_{next}(i, i, d)$, thus either

$$\mathrm{RP}'_i[i, \cdots, B \cup \langle \mathrm{T}, U, d, -, i, seqNum_i, -, -, -, - \rangle, C]$$

$$\xrightarrow{\overline{RREQ_{i,j}}(U,d,dstSeq,i,seqNum_i,\perp,0)} \mathrm{RP}'_i[i, \cdots, B, C]$$

$$\mid \prod_{N_{next}(i,i,d)-j} \overline{RREQ_{i,k}}(U, d, dstSeq, i, seqNum_i, \perp, 0).0$$

     or

$$\mathrm{RP}'_i[i, \cdots, B, C] \mid \prod_{t \in N_{next}(i,i,d)} \overline{RREQ_{i,k}}(U, d, dstSeq, i, seqNum_i, \perp, 0).0$$

$$\xrightarrow{\overline{RREQ_{i,j}}(U,d,dstSeq,i,srcSeq,\perp,0)}$$

$$\mathrm{RP}'_i[i, \cdots] \mid \prod_{N_{next}(i,i,d)-j} \overline{RREQ_{i,k}}(U, d, dstSeq, i, seqNum_i, \perp, 0).0$$

  2. $i \neq s$. This means that i has already received a similar $RREQ$ message, has accepted the route of the message and does not have a valid route to the destination. Moreover, it holds that $\langle s, *, *, j, *, d \rangle \in accepted_i$ and $j \in N_{next}(i, s, d)$, thus either

$$\mathrm{RP}'_i[i, \cdots, B \cup \langle \mathrm{T}, U, d, dstSeq, s, srcSeq, last, hops, prev, e_s \rangle, C]$$

$$\xrightarrow{\overline{RREQ_{i,j}}(U,d,dstSeq,s,srcSeq,last,hops)} \mathrm{RP}'_i[i, \cdots, RT \wr e_s, B, C]$$

$$\mid \prod_{N_{next}(i,s,d)-j} \overline{RREQ_{i,k}}(U, d, dstSeq, s, srcSeq, last, hops).0$$

     or

$$\mathrm{RP}'_i[i,\cdots,B,C] \mid \prod_{t\in N_{next}(i,i,d)} \overline{RREQ_{i,k}}(U,d,dstSeq,s,srcSeq,last,hops).0$$

$$\overline{RREQ_{i,j}}(U,d,dstSeq,s,srcSeq,last,hops) \longrightarrow$$

$$\mathrm{RP}'_i[i,\cdots] \mid \prod_{N_{next}(i,i,d)-j} \overline{RREQ_{i,k}}(U,d,dstSeq,s,srcSeq,last,hops).0$$

Considering the definition of similarity between processes T and P, it is obvious that this action can be followed by any RP_i in P in anyone of the above cases.

Now lets check the receiver $j \in K$. Recall that $\langle s,*,i,*,*,d\rangle \in accepted_j$. We have the following two cases:

1. $\exists\, e \in RT'_j \wedge e{:}d = s$ and $e{:}dstSeq < srcSeq$ or $\nexists\, e \in RT'_j \wedge e{:}d = s$. This means that this request is the first received for the specific flow. Thus

   $$\mathrm{RP}'_j[j,\cdots,B,C] \xrightarrow{RREQ_{i,j}(U,d,dstSeq,s,srcSeq,last,hops)}$$

   $$\mathrm{RP}'_j[j,\cdots,\langle s,srcSeq,\mathrm{T},\infty,\langle i,last,hops{+}1\rangle\rangle \in RT,$$
   $$B \cup \langle \mathrm{T},d,dstSeq,s,srcSeq,last,hops,i,e\rangle, C]$$

2. $\exists\, e \in RT'_j \wedge e{:}d = s$ and $e{:}dstSeq = srcSeq$. This means that j has already received the first $RREQ$ message for this flow, so it contains a route to the source. From the specification we have:

   $$\mathrm{RP}'_j[j,\cdots,B,C] \xrightarrow{RREQ_{i,j}(U,d,dstSeq,s,srcSeq,last,hops)}$$
   $$\mathrm{RP}'_j[j,\cdots,RT \underset{e}{\cup} \langle i,last,hops{+}1\rangle,$$
   $$B \cup \langle \mathrm{F},d,dstSeq,s,srcSeq,last,hops,i,e\rangle, C]$$

By the construction of the restricted system, the loop freedom and link disjointness of the paths in *accepted* sets and the definition of similarity, it is easy to see that in any of the above cases of j, any process RP_j of P will behave in the same way. Thus $P \xrightarrow{a} P'$ and $\langle T',P'\rangle \in \mathcal{R}$.

- If $a = \tau$ and the action has arisen from a communication along a channel of type $RREP$ between processes RP'_i and RP'_j. For process $RP'_i, i \in K$ there are the following possible cases:

  1. $i = d \wedge \langle s,j,*,d)\rangle \in main_i$. In this case i has received the first request message for this flow. Thus

$$\mathrm{RP}'_i[i, \cdots, B \cup \langle \mathrm{T}, U, d, dstSeq, s, srcSeq, last, hops, j, e_s \rangle, C]$$

$$\xrightarrow{\overline{RREP_{i,j}}(d,dstSeq,s,\perp,0)} \mathrm{RP}'_i[i, \cdots, seqNum_i{+}1, \cdots, B, C]$$

or

$$\mathrm{RP}'_i[i, \cdots, B \cup \langle \mathrm{T}, U, d, dstSeq, s, srcSeq, last, hops, j, e_s \rangle, C]$$

$$\xrightarrow{\overline{RREP_{i,j}}(d,dstSeq+1,s,\perp,0)} \mathrm{RP}'_i[i, \cdots, seqNum_i{+}1, \cdots, B, C]$$

2. $j = d \wedge \langle s, *, i, -, i, d \rangle \in accepted_j$. In this case i has received a request message but rejected the new path. Thus

$$\mathrm{RP}'_i[i, \cdots] \mid \overline{RREP_{i,j}}(i, seqNum_i, s, \emptyset, 0).0 \xrightarrow{\overline{RREP_{i,j}}(d,seqNum_i,s,\emptyset,0)} \mathrm{RP}'_i[i, \cdots]$$

3. $i \neq d$ and $e \in RT_i \wedge e\!:\!d = d$ and $e\!:\!Valid = \mathrm{T} \wedge r \in e\!:\!routes$ s.t. $\langle s, *, j, r\!:\!next, r\!:\!last, d \rangle \in accepted_i$. In this case i, an intermediate node, has received a request message to a destination to which it knows a route r. Thus

$$\mathrm{RP}'_i[i, \cdots, B \cup \langle init, d, dstSeq, s, srcSeq, last, hops, j, e_s \rangle, C]$$
$$\xrightarrow{\overline{RREP_{i,j}}(d,dstSeq,s,r:last,e:hops)}$$
$$\mathrm{RP}'_i[i, \cdots, replies \cup \langle s, d, last, r\!:\!last \rangle, \cdots, B, C]$$

4. $i \neq d$ and $e \in RT_i \wedge e : d = s$ and $\exists r \in e : routes$ s.t. $\langle s, *, r : next = j, next, last, d \rangle \in accepted$). In this case i, an intermediate node, has received a reply message that forwards. Thus

$$\mathrm{RP}'_i[i, \cdots, C \cup \langle init, d, dstSeq, s, last, hops, next, e_s \rangle]$$
$$\xrightarrow{\overline{RREP_{i,j}}(d,dstSeq,s,last,e:hops)} \mathrm{RP}'_i[i, \cdots, replies \cup \langle s, d, r\!:\!last, last \rangle, \cdots, C]$$

By the construction of the simplified system and the definition of similarity, it is easy to see that in anyone of the above cases, the action can be followed by any RP_i of P.

Now lets check the receiver $j \in K$. Recall that $\langle s, *, *, i, last, d) \rangle \in accepted_j$. There are two possible cases:

1. $(\exists\, e \in RT \wedge e\!:\!d = d \wedge e\!:\!dstSeq < desSeq) \vee (\nexists\, e \in RT \wedge e\!:\!d = d)$. This means that this reply is the first received for the specific flow. Thus

$$\mathrm{RP}'_j[j, \cdots, B, C] \xrightarrow{RREP_{i,j}(d,dstSeq,s,last,hops)}$$
$$\mathrm{RP}'_j[j, \cdots, e = \langle d, dstSeq, \mathrm{T}, \infty, \langle i, last, hops{+}1 \rangle \rangle \in RT, B,$$
$$C \cup \langle \mathrm{T}, d, dstSeq, s, last, hops, i, e \rangle]$$

2. $\exists\, e \in RT \wedge e{:}d = d \wedge e{:}dstSeq = dstSeq$. This means that this is a duplicate reply of this flow that contains a new loop free and link-disjoint path to the destination. Thus

$$\mathrm{RP}'_j[j, \cdots, B, C] \xrightarrow{RREP_{i,j}(d,dstSeq,s,last,hops)} \mathrm{RP}'_j[j, \cdots, RT \underset{e}{\cup} \langle i, last, hops{+}1\rangle, B, C \cup \langle \mathrm{F}, d, dstSeq, s, last, hops, i, e\rangle]$$

It's obvious that under these conditions, any process RP_j of P will follow this action. Thus $P \xrightarrow{a} P'$ and $\langle T', P'\rangle \in \mathcal{R}$.

This completes the proof. □

This result says that *System can* do the required actions. But this is not enough; it does not guarantee that *System always* exhibit the required behavior. We continue to prove that *System* can do no more than the desired. This is done through a notion of compatible computations of *System* and T_0.

Given a computation $System \xrightarrow{\omega} P$, for each action $start_s(d) \in \omega$, fix sets $Flow(s,d)$ as loop free and link disjoint sets of paths from s to d, and $MainPaths(s,d) \subseteq Flow(s,d)$ as defined earlier.

We say that $T_0 \in \mathcal{T}$ is compatible with this computation if $Flow(s,d) \supseteq Paths(P,s,d)$ and

$$T_0 \stackrel{\text{def}}{=} (\prod_{k \in K} \mathrm{RP}'[k, N, main_k, accepted_k, seqNum_k, replies_k, RT_k, B_k, C_k])$$

where sets $main_k$ and $accepted_k$ are computed from sets $Flow(s,d)$ and $MainPaths(s,d)$ as in the definition of the restricted system, $seqNum_k$ and RT_k are equal to respective $seqNum_k$ and RT_k of *System* and $replies_k = B_k = C_k = \emptyset$.

We now prove that, any set of actions done by *System*, there is a T_0 that can execute a compatible set of actions and lead to similar state.

Lemma 5.2.13 *If* $System \stackrel{\omega}{\Longrightarrow} P$ *then there exists* T_0 *such that,* $T_0 \stackrel{\omega}{\Longrightarrow} T$ *and* P *and* T *are similar.*

Proof. We will prove the result by induction on the length, n, of the transition $System \stackrel{\omega}{\Longrightarrow} P$.

The base case, $n = 0$ is trivially true for any $T_0 \in \mathcal{T}$. Suppose that the result holds for $n = k - 1$ and consider $System \stackrel{\omega}{\Longrightarrow} P' \xrightarrow{a} P$ a transition of length k. Let T_0 be compatible with the computation as defined above. Then, T_0 is also compatible with the computation $System \stackrel{\omega}{\Longrightarrow} P'$ and, by the induction hypothesis, $T_0 \stackrel{\omega}{\Longrightarrow} T'$ where P' and T' are similar. Consider the transition $P' \xrightarrow{a} P$. The following cases exists:

- $a = \tau$ and the internal action took place on a channel that does not belong to any path $w \in Flow(s,d)$ of T_0. Then we may see that for $T = T'$, $T' \stackrel{\epsilon}{\Longrightarrow} T$ and T, P are similar.

- $a = \tau$ and the internal action took place on a channel that belongs to some path $w \in Flow(s,d)$ of T_0. Then using a case analysis similar to the one found in the proof of Lemma 5.2.12 we may find appropriate T such that $T' \stackrel{\tau}{\longrightarrow} T$ and T, P are similar.

- $a = start_s(d)$. In this case we can see that process P depends on whether node s knows a valid path to d or not. In both cases we can find appropriate T such that $T' \stackrel{\tau}{\longrightarrow} T$ and T, P are similar.

- $a = \overline{routeFound_s}(d)$. Suppose that this action was emitted in P' by some process $RP[u, \cdots, C_u]$ such that $\langle \mathrm{T}, d, *, s, *, *, *, * \rangle \in C_u$. Then, it must be that the process has received either a $start_s(d)$ signal and there exists an $e \in RT_u$ s.t. $e : d = d \ \wedge \ e : Valid = \mathrm{T}$ or a message along a channel $RREP_{v,u}(d, dstSeq, s, last, hops), v \in N_u$. In the first case it is easy to find an appropriate T such that $T' \stackrel{\tau}{\longrightarrow} T$ and T, P are similar. In the second case, this implies that either $v = d$ or v knows a valid path to d or it has received a message along channel $RREP_{w,v}, w \in N_v$ and so on. Since the network is connected, this implies that each one of these nodes (except s who first send $RREQ$ messages) have, at some point in the past, received a $RREQ$ message from a new loop free and link disjoint path from s which they accepted. Once accepted this path, a node either replies to the message (if it is the destination node d, or knows a valid path to d), or forwards the message to its neighbors. Due to this sequence of actions, s learns a new path to d and there is an appropriate T such that $T' \stackrel{\tau}{\longrightarrow} T$ and T, P are similar.

□

With the following lemma we complete the proof of Theorem 5.2.1.

Lemma 5.2.14 *$\mathcal{R} = \{(P, Spec_S) \mid S = pending(P)\}$ is a weak bisimulation.*

Proof. Let $(P, Q) \in \mathcal{R}$. First we show that if $P \stackrel{\alpha}{\longrightarrow} P'$ then $Q \stackrel{\alpha}{\longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$. Suppose $P \stackrel{\alpha}{\longrightarrow} P'$. There are three cases:

- $\alpha = start_j(f)$. Then $Q \stackrel{\alpha}{\longrightarrow} Q' = Spec_{S \cup \{\langle j,f \rangle\}}$ and clearly $(P', Q') \in \mathcal{R}$.

- $\alpha = \overline{routeFound_i}(d)$. Then it must be that $\langle i, d \rangle \in S$ and clearly $Q \stackrel{\alpha}{\longrightarrow} Q' = Spec_{S - \{\langle i,d \rangle\}}$ where clearly $(P', Q') \in \mathcal{R}$.

- $\alpha = \tau$. Then we may observe that $pending(P') = pending(P)$ and consequently $(P', Q) \in R$ as required.

We next show that if $Q \xrightarrow{\alpha} Q'$ then $P \xrightarrow{\alpha} P'$ and $(P', Q') \in \mathcal{R}$. Suppose $Q \xrightarrow{\alpha} Q'$. Then there are two cases:

- $\alpha = start_j(f)$ and $Q' = Spec_{S \cup \{\langle j,f \rangle\}}$. Then $P \xrightarrow{\alpha} P'$ where $pending(P') = pending(P) \cup \{\langle j, f \rangle\}$ and clearly $(P', Q') \in \mathcal{R}$.

- $\alpha = \overline{routeFound_i}(d)$ and $Q' = Spec_{S - \{\langle i,d \rangle\}}$. Then it must be that $\langle i, d \rangle \in pending(P)$ and by Lemma 5.2.13 there exists $T_0 \Longrightarrow T$ such that P and T are similar. By Lemma 5.2.10 we know that $T \xrightarrow{\alpha} T'$ and by Lemma 5.2.12 we get that $P \xrightarrow{\alpha} P'$ where $pending(P') = pending(P) - \{\langle i, d \rangle\}$. Thus $(P', Q') \in \mathcal{R}$ which completes the proof.

□

This concludes the proof of Theorem 5.2.1 which shows that algorithm AOMDV delivers the desired behavior, that is every route request will eventually discover at least one path and the paths that are discovered are loop free and link disjoint.

Chapter 6

Conclusions

In this work, we have modeled and analyzed a multipath routing protocol from the area of ad hoc networks using two formalisms, the I/O Automata and the Process Algebra CCS. Specifically, we have modeled the Route Discovery Process of the "Ad hoc on-demand multipath distance vector routing" [36] algorithm, which is an extension of the "AODV" [46] protocol, and we have analyzed the algorithm to prove its correctness. This constitute our main contribution.

We point out that our specifications of AOMDV are significant in themselves since the protocol AOMDV is not given in any formal description. In fact, both of the protocols AOMDV and AODV are given in "natural" language which in many occasions is obscure and can have ambiguous meanings. In addition, the authors of AOMDV in [36] give only the parts that their algorithm extend AODV and not the description of the whole algorithm. This results in missing details in algorithm's description. Instead, the specification of the algorithm make precise these missing and obscure parts helping in the understanding of AOMDV. This illustrates an additional benefit in the use of formal methods in the development and verification of algorithms or systems in general.

Contrasting the specifications and verifications of AOMDV with the two formalisms, the first impression we get is that "they have been developed on different planets" as F. Vaandrager mentions gracefully in [62]. In the same paper the author provides a comparison of the two methodologies regarding their semantics. It is shown that I/O Automata is a subset of Process Algebras. Moving on we see that the two frameworks have many ideas in common, but they have also many differences.

6.1 Comparison of I/O Automata and Process Algebras

Beginning with the specification of AOMDV in each framework, we can identify the first differences. Process Algebras have many operators that give precise meaning to the states and operations of the system. Moreover, a new operator can be easily added if needed. These operators are used to compose simple processes to complex systems. In contrast, I/O Automata have only a few operators such as composition which composes the automata to built systems. A state (set of variables) of an I/O Automaton is central and has global scope in the automaton specification. In contrast in Process Algebras the parameters of the process state has to be carried around and their scope is limited to the actions in the specific process. I/O Automata focus on the description of the actions that each component does while Process Algebra focus on the states that a system can be. Thus, I/O Automata are closer to the development way of thinking than Process Algebra, since the development way of thinking concentrates on what each component has to do rather than on what state the system should reach after a certain action.

Another main difference is that each action in I/O Automata is given in terms of precondition-effects. In this way each action is enabled when its preconditions are fulfilled and the enabled actions can be executed in nondeterministic manner. Recall that input actions have no preconditions, thus they are always enabled. In contrast, in Process Algebras, the actions of each process are structured by the semantic operators and they are enabled only if they reach the top of the structure. Moreover, the enabled actions can be executed in a nondeterministic manner, just like in I/O Automata.

Each action in I/O Automata is an atomic action that consists of a series of state changes. In contrast, each action in Process Algebra changes the state of the system only once, probably though changing more than one parameters. Moreover, this difference of action granularity affects the proofs of each formal method. In Process Algebra the proof has to include more actions than in I/O Automata, where the proof is limited to fewer actions which are split to more cases of the same action.

Both specifications of I/O Automaton and Process Algebras can become more abstract or more detailed depending on the goal of the study.

Concerning the proofs of each formalism, I/O Automata split the required behavior in small invariant assertions and safety lemmas that prove that the system does not do anything that is not supposed to do. In addition, some liveness lemmas show that even if everything is ok and fair in the system, it will eventually deliver

the required behavior. In contrast, Process Algebras uses simulation and bisimulations results to conclude that the system works as it is supposed to work. In the proof of AOMDV we used the following method: We built a restricted system that exhibits the desired behavior and we proved that the general system is a simulation of the restricted. In other words we proved that the general system can do the "good" things. Moreover, we proved bisimilarity to the system specification, proving that not only it can do the "good" things, but it will do them. Thus, I/O Automata concentrate to preclude any wrong behavior while Process Algebra concentrates to show that the desired behavior is the only behavior that the system can have.

This observation leads to an important similarity of the two frameworks. Both formalisms provide abstract descriptions of systems and algorithms, thus they don't take into account faulty situations as crashes, message errors and losses unless they are modeled in the specification. In other words if the goal of the analysis is to determine the behavior of the system or the algorithm in study in such situations, then these cases have to be modeled in the specification.

In general, I/O Automata have local focus, that is they focus on the actions of each automaton individual. This makes it easier to proof local statements than global statements. Instead, in Process Algebra, the focus of the proof can be easily transfered from global focus to local focus thus the proofs for global and local statements have the same degree of difficulty.

6.2 Future work

This dissertation investigates the foundations of the modeling, development and analysis of ad hoc networks through the modeling and analysis of an algorithm of ad hoc networks. In specific, this work regards the route discovery phase of the algorithm AOMDV, a multipath routing algorithm of ad hoc networks. The algorithm was modeled and analyzed by two formalisms, the I/O Automata and Process Algebras.

As a next step in this work, the second phase of the same algorithm, the maintenance phase, will be modeled and analyzed to check that the correctness of the algorithm is not affected by this phase. In addition, the algorithm will be checked applying timeouts on the created paths.

Moreover, in this work we give a first comparison of the two formalisms for their applicability in the field of ad hoc networks. This work can be used as a reference point for future work including further extensive study of the two formalisms in the

field of ad hoc networks. This can be achieved by modeling and analyzing more algorithms, or the same algorithm in more dynamic environment modeling node movements and failures. This will lead to a deeper investigation of the strengths and weaknesses of each formal method and possibly one of them will surpasses the other or a new hybrid formalism will be born that includes the strengths of both formalisms.

Bibliography

[1] The IOA Toolkit. http://theory.lcs.mit.edu/tds/ioa/.

[2] G.-S. Ahn, A.T. Campbell, A. Veres, and L.-H. Sun. SWAN: Service Differentiation in Stateless Wireless Ad Hoc Networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2002*, volume 2, pages 457– 466, 2002.

[3] P. Attie and N. Lynch. Dynamic Input/Output Automata: a Formal Model for Dynamic Systems. In *Proceedings of the 12th International Conference on Concurrency Theory*, pages 21–24, 2001.

[4] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the 19th annual ACM conference on Theory of computing, STOC '87*, pages 230–240, 1987.

[5] R. Baldoni, A. Virgillito, and R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. In *Proceedings of the 7th International Symposium on Computers and Communications, ISCC 2002*, pages 539– 544, 2002.

[6] M. Bechler, H.-J. Hof, D. Kraft, F. Pahlke, and L. Wolf. A cluster-based security architecture for ad hoc networks. In *Proceedings of the 23rd AnnualJoint Conference of the IEEE Computer and Communications Societies, INFOCOM 2004*, volume 4, pages 2393– 2403, 2004.

[7] M. Benchaiba, A. Bouabdallah, N. Badache, and M. Ahmed-Nacer. Distributed mutual exclusion algorithms in mobile ad hoc networks: an overview. *SIGOPS Operating Systems Review*, 38(1):74–89, 2004.

[8] J. A. Bergstra. *Handbook of Process Algebra*. Elsevier Science Inc., 2001.

[9] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of ACM*, 49(4):538–576, 2002.

[10] L. Cardelli and A. Gordon. Mobile Agents. In *Proccedings of the International Conference on Foundations of Software Science and Computation Structures, FOSSACS'98*, LNCS 1378, pages 140–155, 1998.

[11] S. Chiyangwa and M. Kwiatkowska. A Timing Analysis of AODV. In *Proceedings of the 7th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, pages 306–322, 2005.

[12] I. Chlamtac, M. Conti, and J. Liu. Mobile Ad Hoc Networking: Imperatives and Challenges. *Ad Hoc Networks*, 1(1):13–64, 2003.

[13] S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. A. Shvartsman, and J. Welch. Virtual Mobile Nodes for Mobile Ad Hoc Networks. In *Proceedings of the 18th International Symposium on Distributed Computing*, pages 230–244, 2004.

[14] S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. Welch. GeoQuorums: Implementing Atomic Memory in Ad Hoc Networks. In *Proceedings of 17th International Symposium on Distributed Computing*, pages 306–320, 2003.

[15] S. Dolev, S. Gilbert, E. Schiller, A. A. Shvartsman, and J. Welch. Autonomous Virtual Mobile Notes. In *Proceedings of the 3rd Annual ACM/SIGMOBILE International Workshop on Foundations of Mobile Computing*, 2005.

[16] C. Ene and T. Muntean. A Broadcast-based Calculus for Communicating Systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium, IPDPS '01*, pages 1516–1525, 2001.

[17] L.M. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2001*, volume 3, pages 1548–1557, 2001.

[18] W. Fokkink. *Introduction to Process Algebra*. Springer-Verlag New York, Inc., 2000.

[19] C. L. Fullmer and J. J. Garcia-Luna-Aceves. Solutions to hidden terminal problems in wireless networks. *SIGCOMM Computer Communication Review*, 27(4):39–49, 1997.

[20] S. Giordano. *Handbook of Wireless Networks and Mobile Computing*, chapter Mobile Ad-Hoc Networks. John Wiley and Sons, 2002.

[21] J. C. Godskesen. A calculus for mobile ad hoc networks. In *Proceedings of Coordination'07 to appear*, 2007.

[22] K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas, and R. B. Tan. Fundamental control algorithms in mobile networks. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '99*, pages 251–260, 1999.

[23] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32, 1985.

[24] M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.

[25] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In *Proccedings of the 3rd International Workshop on High-Level Concurrent Languages*, pages 3–17, 1998.

[26] A.D. Jaggard and V. Ramachandran. Relating two formal models of path-vector routing. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2005*, volume 1, pages 619– 630, 2005.

[27] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O Automata: A Mathematical Framework for Modeling and Analyzing Real-Time Systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 166–177, 2003.

[28] N. Li and J.C. Hou. Topology control in heterogeneous wireless networks: problems and solutions. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2004*, volume 1, page 243, 2004.

[29] N. Li, J.C. Hou, and L. Sha. Design and analysis of an MST-based topology control algorithm. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2003*, volume 3, pages 1702– 1712, 2003.

[30] N. Lynch, R. Segala, and F. Vaandrager. Compositionality for Probabilistic Automata. In *Proceedings of the 14th International Conference on Concurrency Theory*, pages 208–221, 2003.

[31] N. Lynch, R. Segala, and F. Vaandrager. Hybrid I/O Automata. *Information and Computation*, 185(1):105–157, 2003.

[32] N. Lynch and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[33] N.A. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[34] N. Malpani, J. L. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications, DIALM '00*, pages 96–103, 2000.

[35] M. Marina and S. Das. On-Demand Multi-Path Distance Vector Routing in Ad Hoc Networks. In *Proceedings of the 9th International Conference on Network Protocols, ICNP'01*, page 14, 2001.

[36] M. K. Marina and S. R. Das. Ad hoc on-demand multipath distance vector routing. *Wireless Communications and Mobile Computing*, 6(7):969 – 988, 2006.

[37] M. Merro. An observational theory for mobile ad hoc networks. In *Proceedings of the 23rd Conference on the Mathematical Foundations of Programming Semantics (MFPS XXIII)*, volume 173 of *Electronic Notes in Theoretical Computer Science*, pages 275–293, 2007.

[38] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[39] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[40] R. Milner. The Poyadic π–Calculus: a Tutorial. In *Logic and Algebra of Specification*, pages 203–246. 1991.

[41] K. Nakano and S. Olariu. Randomized Leader Election Protocols for Ad-hoc Networks. In *Proceedings of the 7th International Colloquium on Structural Information and Communication Complexity, SIROCCO*, pages 253–267, 2000.

[42] S. Nanz and C. Hankin. Static analysis of routing protocols for ad-hoc networks. In *Proceedings of the 5th ACM SIGPLAN and IFIP WG 1.7 Workshop on Issues in the Theory of Security, WITS04*, pages 141–152, 2004.

[43] S. Nanz and C. Hankin. A framework for security analysis of mobile wireless networks. *Theoretical Computer Science*, 367(1):203–227, 2006.

[44] A. Nasipuri, R. Castaneda, and S.R. Das. Performance of Multipath Routing for On-Demand Protocols in Mobile Ad Hoc Networks. *Mobile Networks and Applications*, 6(4):339–349, 2001.

[45] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: theory and application. *Information and Computation*, 114(1):131–178, 1994.

[46] C. Perkins, E. Belding-Royer, and S. Das. RFC3561: Ad hoc On-Demand Distance Vector (AODV) Routing, 2003.

[47] C. E. Perkins. *Ad Hoc Networking*. Addison-Wesley, 2001.

[48] C. E. Perkins and P. Bhagwat. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. *SIGCOMM Computer Communication Review*, 24(4):234–244, 1994.

[49] A. Philippou and G. Michael. Verification Techniques for Distributed Algorithms. In *OPODIS*, pages 172–186, 2006.

[50] Y. Qiu and P. Marbach. Bandwidth Allocation in Ad Hoc Networks: a Price-Based Approach. In *Proceedings of 22nd Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM 2003*, volume 2, pages 797 – 807, 2003.

[51] L. Ramachandran, M. Kapoor, A. Sarkar, and A. Aggarwal. Clustering algorithms for wireless ad hoc networks. In *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications, DIALM '00*, pages 54–63, 2000.

[52] R. Ramanathan and R. Rosales-Hain. Topology control of multihop wireless networks using transmit power adjustment. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2000*, volume 2, pages 404–413, 2000.

[53] E.M. Royer and C.-K. Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, 6(2):46–55, 1999.

[54] A. Savvides, C.-C. Han, and M. B. Strivastava. Dynamic fine-grained localization in Ad-Hoc networks of sensors. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking, MobiCom '01*, pages 166–179, 2001.

[55] P. Sewell. Global/Local Subtyping and Capability Inference for a Distributed π-calculus. In *Proccedings of the 25th International Colloquium in Automata, Languages and Programming*, LNCS 1443, pages 695–706, 1998.

[56] P. Sewell, P. Wojciechowski, and B. Pierce. Location Independence for Mobile Agents. In *Proceedings of the International Conference on Computer Languages*, volume 1686, 1998.

[57] V. Srinivasan, C.F. Chiasserini, P. Nuggehalli, and R.R. Rao. Optimal rate allocation and traffic splits for energy efficient routing in ad hoc networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2002*, volume 2, pages 950– 957, 2002.

[58] K. Sundaresan, R. Sivakumar, M.A. Ingram, and T.-Y. Chang. A fair medium access control protocol for ad-hoc networks with MIMO links. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2004*, volume 4, pages 2559– 2570, 2004.

[59] L. Tassiulas and S. Sarkar. Maxmin fair scheduling in wireless networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2002*, volume 2, pages 763– 772, 2002.

[60] Y.-C. Tseng, C.-S. Hsu, and T.-Y. Hsieh. Power-saving protocols for IEEE 802.11-based multi-hop ad hoc networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2002*, volume 1, pages 200–209, 2002.

[61] Y.-C. Tseng, S.-L. Wu, W.-H. Liao, and C.-M. Chao. Location awareness in ad hoc wireless mobile networks. *IEEE Computer*, 34(6):46–52, 2001.

[62] F. Vaandrager. On the Relationship Between Process Algebra and Input/Output Automata. In *Proceedings of the 6th Symposium on Logic in Computer Science*, pages 387–398, 1991.

[63] J. E. Walter, J. L. Welch, and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Network*, 7(6):585–600, 2001.

[64] R. Wattenhofer, L. Li, P. Bahl, and Y.-M. Wang. Distributed topology control for power efficient operation in multihop wireless ad hoc networks. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2001*, volume 3, pages 1388–1397, 2001.

[65] Y. Xue, B. Li, and K. Nahrstedt. Optimal resource allocation in wireless ad hoc networks: a price-based approach. In *IEEE Transactions on Mobile Computing*, volume 5, pages 347– 364, 2006.

[66] Z. Ye, S.V. Krishnamurthy, and S.K. Tripathi. A framework for reliable routing in mobile ad hoc networks. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2003*, volume 1, 2003.

[67] M. G. Zapata and N. Asokan. Securing ad hoc routing protocols. In *Proceedings of the 3rd ACM workshop on Wireless security, WiSE '02*, pages 1–10, 2002.