Imperial College London
Department of Computing

Fault Localization in Service-Based Systems hosted in Mobile Ad Hoc Networks

Petr Novotny

July 2013

Supervised by Alexander L. Wolf

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Throughout the text of this thesis is used the term “We” referring to the author of the work described in this thesis. Although the work was carried out solely by the author, it was under the supervision and advice of Alexander L. Wolf and with additional advice and guidance from Bong Jun Ko.

Petr Novotny

Dedication

I would like to dedicate this Doctoral thesis to my parents, Miloš and Jiřina. I would not have contemplated this road if not for the inspiration I have received early in my life from my father who instilled within me the passion for science and technology as well as love of creative pursuits. At the same time, only the love, continued care, support and encouragement of my mother allowed me to endeavor in this process. To my parents, thank you.

Acknowledgements

Completing my PhD degree is without a doubt the most challenging activity of my life to date. The best and worst moments of my doctoral journey have been shared with many people. It has been a great privilege to spend several years in the Department of Computing at Imperial College London, and people I have met here will always remain dear to me.

First and foremost I wish to thank my advisor, professor Alexander L. Wolf, for all the academic as well as personal support I have received from him. His friendly attitude and sense of humor has been of great help in overcoming many of the difficulties which arose throughout my doctoral training. The transformation one has to go through to become a researcher is not an easy one, however, with his enthusiasm, his inspiration, and his great efforts to explain things clearly and simply, he helped to make the process fun for me. Throughout my training, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas. I would have been lost without him.

A special thank goes to Bong Jun Ko, for his unflagging friendly support and for always keeping his door open. Without his help my work would not be of the same quality. He has been a strong and supportive guide, source of knowledge as well as role model to me throughout my doctoral training, his guidance has served me well and I owe him my heartfelt appreciation.

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

Related Publications

Publications arising from work in this thesis:

Novotny, P. and Wolf, A.L. and Bong Jun Ko. Fault Localization in MANET-Hosted Service-Based Systems. 2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS) 243 - 248.

Novotny, Petr and Wolf, Alexander L. and Ko, Bong Jun. Discovering Service Dependencies in Mobile Ad hoc Networks. 2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013) 527 - 533.

Abstract

Fault localization in general refers to a technique for identifying the likely root causes of failures observed in systems formed from components. Fault localization in systems deployed on mobile ad hoc networks (MANETs) is a particularly challenging task because those systems are subject to a wider variety and higher incidence of faults than those deployed in fixed networks, the resources available to track fault symptoms are severely limited, and many of the sources of faults in MANETs are by their nature transient.

We present a suite of three methods, each responsible for part of the overall task of localizing the faults occurring in service-based systems hosted on MANETs. First, we describe a dependence discovery method, designed specifically for this environment, yielding dynamic snapshots of dependence relationships discovered through decentralized observations of service interactions. Next, we present a method for localizing the faults occurring in service-based systems hosted on MANETs. We employ both Bayesian and timing-based reasoning techniques to analyze the dependence data produced by the dependence discovery method in the context of a specific fault propagation model, deriving a ranked list of candidate fault locations. In the third method, we present an epidemic protocol designed for transferring the dependence and symptom data between nodes of MANET networks with low connectivity. The protocol creates network wide synchronization overlay and transfers the data over intermediate nodes in periodic synchronization cycles.

We introduce a new tool for simulation of service-based systems hosted on MANETs and use the tool for evaluation of several operational aspects of the methods. Next, we present implementation of the methods in Java EE and use emulation environment to evaluate the methods. We present the results of an extensive set of experiments exploring a wide range of operational conditions to evaluate the accuracy and performance of our methods.

Contents

List of Tables

List of Figures

1. Introduction

Mobile ad hoc networks (MANETs) are used in difficult or remote situations, such as emergency response or forest-fire fighting, where a reliable, fixed-network communication infrastructure may be absent. However, the applications deployed upon MANETs are increasingly expected to exhibit sophisticated features, mimicking the availability of rich applications in "normal" network environments.

It is no surprise, then, that MANETs are now hosting applications that are architected as flexible and dynamic collections of *software services*. In *service-based systems*, such as those based on the Service-Oriented Architecture (SOA) or Web Services frameworks, computations are structured as sets of services that respond to requests, where a request typically originates at a user-facing client. The computation required to fulfill each request results in a cascade of further requests across some subset of the services; services make requests on other services in order to fulfill the requests made upon them. The set of messages exchanged during the processing of a single client request is referred to as a *conversation*.

Our work is carried out in a context of a defense research project. Hence, we primarily focus on dependability of service-based systems supporting operations of army units on missions in terrain conditions as well as on emergency response missions such as search and rescue and forest firefighting. The goal of our work presented in this thesis is to allow localization of faults experienced by members of mobile units while using mobile computer devices with client applications of service-based systems hosted in the MANETs.

Fault Localization

We are concerned with a crucial operational aspect of service-based systems deployed on MANETs: *fault localization* (also called *fault identification*). This is a particularly challenging management task because the systems are subject to a wider variety and higher incidence of faults than those deployed in fixed networks, the resources available to track fault symptoms are severely limited, and many of the causes of failures in MANETs are by their nature transient (e.g., a mobile host that temporarily moves out of radio range).

Further complicating the situation is that faults at the network level may not manifest themselves as failures at the service level, since some of those faults may occur outside of the time during which the relevant services are communicating, or involve communication outside of the relevant faulty conversation. Moreover, service-based systems are designed to mask certain kinds of faults through mechanisms for dynamic service discovery and (re)binding. Therefore, a fault localization method suitable for MANET-hosted service-based systems must be adept at sifting through noisy data.

We have designed fault localization method that makes use of two kinds of basic information: (1) a service *dependence graph* rooted at the client that initiated the conversation and reported the failure, and (2) various network- and service-level *fault symptoms* recorded in host-local logs. The dependence graph is discovered at run time by a decentralized monitoring system deployed in the network. The monitoring system also collects the symptoms from the local logs. The graph and symptoms are used to carry out a centralized fault propagation analysis based on a *fault propagation pattern*, which indicates how faults can be propagated through the system from root causes to clients. The result of the analysis is a *fault propagation model* that relates possible root causes to the failure reported by the client. Finally, the faults are *ranked* for their likelihood of causing the failure.

Existing fault localization methods designed for the MANET environment [14, 18, 22, 23, 26, 47, 48] focus on network-level identification of individual faulty hosts and/or the links between them. The methods are limited to low-level observations and measurements of packet flows and host failures, and therefore are blind to the end-to-end context of the service-level conversations affected or unaffected by those faults. Our fault localization method instead makes use of both service- and network-level information. This enables a decoding of the fault propagation through the hosted services via the network. The client is therefore a good entry point for fault localization, tracing failures back to their sources using information that is unavailable from observations at the network level alone.

Dependence Discovery

A critical aspect of our fault localization method is understanding the dependencies among the services of the service-based system. The importance of dependence information increases with the complexity of the system, both in terms of the number of interacting components required to carry out a given computation and the nature of the environment in which the system operates.

Obtaining dependence information in the service-based system is made difficult by the inherent loose coupling of services, as many dependencies are unknown at design time, and only established at run time through a dynamic service binding mechanism (so-called "service discovery"). The consequence is that the dependencies among run-time instances of services are not something that can be reliably specified before execution, but instead must be discovered during or after execution.

Existing dependence discovery methods focus on statically structured systems operated in fixed networks [5, 6, 7, 8, 9, 17, 19, 43, 53, 66]. The motivation for these methods is the absence of an accurate specification of dependencies, so a map of those dependencies must be discovered by observing the executing system. A critical assumption made by these methods is that the dependence data, although changing, is relatively stable over time. The significance of the stability assumption is that the methods can make use of statistical techniques based on data collected over long execution periods. Furthermore, by operating in the context of a fixed-network environment, the methods can assume no practical limits on the storage, computational, and communication resources needed to support those statistical techniques.

The context for our work is instead service-based systems deployed on mobile ad hoc networks (MANETs). Mobility and ad hoc networking bring increased dynamicity to service dependencies, beyond those caused by the basic service-binding regime of SOA or Web Services. Moreover, the MANET environment is typically characterized by severe limitations on the resources available for dependence discovery. Existing methods based on the stability assumption cannot adequately cope with such high levels of dynamicity nor stringent resource constraints.

We have formulated a dependence discovery method that is more suitable for services operating in the challenging MANET environment. Our intuition is that dependence discovery must be focused on capturing snapshots of dependence data relevant to each service request of concern, rather than on the tradition of determining statistical averages for long-term, system-wide dependencies as a whole. Furthermore, the method must be lightweight in its resource usage, which to our thinking means that dependence data should be collected locally, aggregated locally, and drawn to some central location only when and if needed to carry out the fault localization tasks.

Our approach is based on the use of *monitors* deployed onto the mobile hosts. The monitors collect dependence data by observing the message traffic between services and extracting relevant information. The data collected by the monitors provide only a local view of the dependence information. When a *dependence graph* capturing the dependence relationships among the services is required to carry out fault localization, the monitors are contacted by a central discovery element charged with integrating the data. Importantly, only the monitors relevant to a particular analysis question typically need to be contacted, and therefore communication can be reduced. Moreover, the monitors can aggregate the data they collect, and can impose limits on the amount of data they store.

In order to employ our dependence discovery method, three basic prerequisites must be met. First, to obtain complete dependence information, the monitors should be deployed on the mobile hosts that are either the source or the target of service messages; intermediate hosts in the network used only to store and forward network-level messages are not involved in data collection. Second, the monitors need access to synchronized clocks to allow consistent time-stamping of the collected dependence data. Clock synchronization in MANETs is a well-researched topic, with techniques available to achieve precision of tens or even single microseconds [45, 68] The shortest time period we use to time stamp aggregated data is 6 milliseconds, well within this precision. Third, the monitors must be able to observe service messages and obtain certain information from those messages, including such things as client and service identifiers. On the other hand, there is no need for the monitors to have access to the payload of messages. This kind of general information is typically available and visible, since it is used by the underlying service infrastructure to manage service interactions.

Distributed Data Harvesting

The distinguishing aspect of the MANET environment is the limited connectivity between network nodes induced by the dynamism of the network topology. In traditional networks end-to-end path needs to be established before communication can occur. However, networks with dynamic topology are characterized by lack of these paths. The end-to-end path requires series of links between two nodes. In wireless networks such as MANETs, the quality of links is affected by behavioral properties of the nodes (such as nodes getting in and out of range of other nodes), by the nodes resources (such as battery life and transmit power), and by the link properties (such as interference and noise). The links thus can be connected or disconnected at any time.

Ad hoc routing protocols enable network communication in the dynamic environment, however, they have certain limitations and they cannot overcome the limits of the network links. Thus, the network may temporarily partition into several segments with reduced or nonexistent connectivity between them. Moreover, currently used ad hoc routing protocols such as OLSR, have certain limitations which make routing over multiple links unreliable (i.e. latency in synchronization of topology database or disparities in data rates used in neighbor discovery and actual data traffic) [40].

The limited connectivity is further exaggerated in runtime analysis of service-based systems deployed on MANETs. The fault localization and the dependence discovery methods require access to the monitoring data of nodes hosting services involved in the analyzed conversation. However, the required nodes might be several hops away from nodes collecting the monitoring data and thus the connectivity might be insufficient. Moreover, the collection of the data may start sometime after the conversation ended and thus the physical topology may have changed and further decreased the connectivity.

Several distributed algorithms and delay-tolerant networking protocols were proposed for the transfer of data in networks with limited connectivity. However, the methods either do not increase availability of the data sufficiently such as data replication techniques [36, 28, 29, 30, 11, 10, 50, 46, 44, 56, 65, 35, 32] or the use of distributed hash table [34, 49, 41, 38, 21, 42], or they lack the capacity to maintain consistency and integrity of the transferred data such as existing epidemic protocols [44, 55, 67]

We have designed new type of *epidemic protocol* with several distinctive features, suitable for transferring monitoring data between MANET nodes with low connectivity. The algorithm uses “gossiping” to create a network wide data synchronization overlay. Hence, instead of attempting to pass data on direct end-to-end paths between source and destination nodes, the data are transferred over intermediate nodes in successive synchronization cycles.

On nodes hosting services, monitoring data are continuously collected as the services are processing clients’ requests. The fault localization and dependence discovery methods require access into this continuous data in order to extract occurrences of dependencies and symptoms relevant to the analyzed conversation. Our method transfers data from the nodes hosting services into backup stores on nodes analyzing conversations. The fault localization and dependence discovery methods then access the local backup stores instead of to request the data on-demand from the remote monitors. The method is based on the use of *synchronization agents* deployed onto the mobile hosts. The synchronization agents repeatedly in cycles synchronize with neighbor *peers* incremental changes in the monitoring data. The selection of the peers is designed to take into account the peculiarities of the dynamic topology and selects nodes based on connectivity metrics. The agents keep track of past synchronizations with each peer and in successive cycles only accumulated changes are passed on. In each cycle, all new locally collected data as well as all new data received from neighbor peers are synchronized. The method imposes limits on age of the transferred data to minimize the network overhead of the method. Moreover, the transferred data can be aggregated to further reduce the network overhead.

This thesis makes the following specific contributions:

- A dependence discovery method
  - that allows the engineer to trade accuracy against cost;
  - yielding probabilistically accurate dependence graphs.
- A fault localization method
  - to synergistically analyze fault symptom data gathered at both the service and network levels;
  - a fault propagation model based on propagation patterns devised specifically for the operational environment;
  - timing-based and probabilistic root-cause ranking algorithms.
- A distributed data harvesting method
  - a delay-tolerant algorithm;
  - to transfer data between MANET nodes without direct connectivity;
- We have developed experimental toolset to validate our methods
  - a simulator that closely replicates behaviors of service-based systems running on MANETs. The simulator is an extension to popular accurate packet based network simulator.
  - generic Web Service system implemented in Java EE, designed for analysis of service-based systems.
- An extensive, experimental analysis
  - an simulation-based, experimental analysis of dependence discovery and fault localization methods;
  - an implementation of all of our methods in Java EE; and
  - an emulation-based, experimental analysis of the implemented methods.

Clearly, our dependence discovery and fault localization methods are subject to inaccuracies due to effects such as the delay between data collection and data analysis, storage constraints that might require monitors to "forget" some data, failures in hosts and links, and inaccuracies in the discovered dependencies representing the system structure. The essence of our work is to understand how these and other factors impact the accuracy of our dependence discovery and fault localization methods, subject to various tuning and environmental parameters. Moreover, we study how accuracy of the produced dependence graphs impacts accuracy of the fault localization.

In this theses we introduce a new simulation tool for service-based systems hosted on MANETs. With the system and behavioral models of services built on top of a packet-based simulator, our approach allows the replication of various critical aspects, such as the cascading flows of messages in complex conversations, comprehensive client-driven workload profiles, and the propagation of faults through services. Furthermore, the simulator provides generic and easily extended models that can be used to capture modern service-based platforms, such as SOA, operating in MANET or hybrid networks.

We carry out the evaluation of our methods in two series of experiments. In the first series, based on our simulation tool for service-based systems, we focus on the dependence discovery and fault localizations methods and evaluate their sensitivity to a distinguishing aspect of the MANET environment, namely *time-dependent behavior*. The evaluation is carried out through a series of simulation-based experiments under various scenarios that represent a range of mobile-network dynamics, service connectivity and dynamics, and critical parameter settings. Because no benchmarks yet exist for MANET-hosted service-based systems, we develop synthetic data to explore the space of independent variables. In evaluation of the dependence discovery, the dependent variables in these experiments are the *true positives* and *false positives* in the discovered dependence relationships. In evaluation of the fault localization, the primary dependent variable is the *mean position of correct fault* in the ranked list of candidates.

In the second series of experiments, we focus on the distributed data harvesting method and its capacity to transfer monitoring data between nodes. We provide implementation of our methods in Java EE and we evaluate the implemented methods in emulation-based experiments. The evaluation centers on impact of availability of the data transferred by the distributed data harvesting method on accuracy of the dependence discovery and the fault localization methods.

The thesis is structured as follows. We next review prior and related work in Chapter 2.

In Chapter 3 we present our dependence discovery method and its underlying dependence model. In Chapter 4 we describe our fault localization method, including its fault propagation model and ranking algorithms. In Chapter 5 we present our distributed data harvesting method as a means of transferring of monitoring data within MANETs, required by the dependence discovery and fault localization methods. In Chapter 6, we describe in detail the experimental tools we have used to evaluate our methods. Moreover, we also describe the implementation of our methods in Java EE. We then present our experimental evaluation of the methods in Section 7. Finally, we conclude with a summary of the thesis and a look at on-going and future work in Section 8.

2. Related Work

In this chapter we review the related work. We begin by looking at the techniques of the dependence discovery forming the basis of our fault localization method by providing a model of a system. Next, we look at the existing fault localization approaches designed for distributed systems. At the end, we review methods used in transferring of data across networks with dynamic topology.

2.1. Dependence Discovery

Existing dependence discovery methods can be generally classified as to whether they operate at the network level or at the (application) service level. Network-level discovery [5, 6, 7, 8, 19, 43] focuses on coarse-grained dependencies between network hosts. Network-level dependencies are usually described in terms of IP addresses and port numbers. They can be augmented with additional information, such as port mappings [17] or a classification of client applications [53]. On the other hand, service-level discovery [9, 16, 66, 15] focuses on the detection of fine-grained relationships between services. Services are hosted in application containers, such as J2EE and .NET, and typically associated with application identifiers, such as URLs.

Our dependence discovery method, although informed by the network level, operates at the service level in the sense that we wish to discover service dependencies that can be used for fault localization in service-based software systems. In particular, we focus on discovering service-level dependencies in MANETs, where depedencies may change at a rapid rate. It is important to point out that we do not assume the availability of prior information, such as a port mapping, application categories, or even a specification of the services, nor do we require changes in existing software components to support discovery as found in other methods [8, 16].

Many of the service-level discovery methods apply statistical techniques to traffic traces collected by network hosts and monitors. The statistical techniques correlate network packets or service-level messages and identify co-occurrences of messages across different services. While the particular statistical techniques may differ (e.g., correlation based on a time window [5, 6, 7], delay distribution [17], time difference of messages [9], or timing and frequency of packet flows [19]), all of the methods share the same limitation: they require a long period of time to collect statistically stable data and, therefore, are inappropriate in highly dynamic environments such as MANETs.

Alternatives to statistical approaches do exist. For example, in Macroscope [53] a subset of packets and network connections is sampled and analyzed to identify relationships between network flow data and applications. Lu et al. [43] collect system log data and correlate the events in the logs using data-mining techniques. Magpie [8] instead correlates the events based on input from (human) network operators. These methods, however, require the transfer of large amounts of trace data from the collection points to a central analysis element, which is prohibitive in resource-constrained MANETs. In contrast, our method transfers dependence information selectively and on demand, contacting only the monitors that are potentially relevant to the events of interest (e.g., the possible receivers of a failed service request). In addition, the monitors actively summarize and aggregate dependence information before that information is transferred to the analysis node (e.g., they may transfer only the number of messages exchanged between two services, rather than sending the content of those messages).

Pinpoint [16] uses a technique for correlating messages that is similar to the one found in our method. The technique makes use of identifiers to uncover flows of end-to-end processing through servers in a fixed network. Pinpoint requires application servers to insert the identifiers into the headers of messages sent over extensible protocols such HTTP or SOAP. The identifier is passed in processing threads within components and inserted into messages sent in the flow of processing. Monitors deployed in the network record requests into traces from which the paths are reconstructed, post hoc, and further aggregated into dependencies. Their technique is a general one, not restricted to service-based systems.

Similarly, we make use of what are called *conversation identifiers* in service-based systems. In service-based systems, a conversation is the set of all messages exchanged during the processing of a single client request. Typically, the conversation is identified across the messages using tags inserted into message headers; the tags contain a unique conversation identifier. Of course, conversation identifiers must be implemented by service programmers. Fortunately, this is a relatively routine task, as there are several existing standards for doing so, including WS-Addressing,[1] WS-SecureConversation,[2] and WS-Coordination,[3] that are increasingly used in current-day applications. We assume here that conversation identifiers are available for inspection by our monitors.

2.2. Fault Localization

Although fault localization has been studied extensively for wired network environments (see Steinder and Sethi [60] for a survey), attempts to solve the problem for wireless networks are only more recent. Some of these newer approaches address the issue in the context of stationary, infrastructure-based wireless networks [1, 52, 54], but are not able to handle the dynamics of mobile ad hoc networks.

The few existing approaches to fault localization in MANETs can be generally classified into two broad categories.

In the first category fall those seeking to identify a set of fault-free hosts, for which methods based on *output comparison* have been proposed. In the basic method, pre-defined tasks are assigned to network hosts and their outputs are compared to ones defined *a priori*; a hosts is considered fault free if the results match. This approach was introduced into MANETs by Chessa and Santi [18]. In their approach, hosts continuously diagnose the status of their immediate neighbors and learn the status of all other hosts in the network via dissemination protocols. Variations and improvements, using different task assignment strategies and/or state dissemination techniques, have also been developed [22, 23]. Because these approaches focus on the identification of fault-free hosts through continuous monitoring and dissemination, they are indirect (and inefficient) methods for fault localization.

In the second category, more relevant to the subject of this thesis, are methods seeking to identify individual faulty hosts and links in MANETs. Natu and Sethi [47, 48] use a temporal event-correlation approach with dynamic discovery of the network topology. Their method periodically identifies the end-to-end paths and the states of hosts based on correlations of measurements, identifying faulty hosts and links through active probing of paths. Fecko and Steinder [26] use a combinatorial approach and variance analysis to correlate the occurrences of multiple failures in a network. Their approach is based on probabilistic dependence graphs obtained from expert knowledge of the intended system structure and

[1]`http://www.w3.org/2002/ws/addr/`
[2]`http://www.ibm.com/developerworks/library/specification/ws-secon/`
[3]`http://www.ibm.com/developerworks/library/specification/ws-tx/`

the history of system faults. Cavalcante and Grajzer [14] propose a form of probabilistic fault propagation model based on Bayesian probability for a specific architectural model of dependable networks [63]. All these methods are limited by the fact that they focus on identifying the faults at low-level network components (hosts and links), not taking into account the applications actually making use of the MANET. Because network-level faults do not necessarily manifest themselves as failures in the application layer and (obviously) vice versa, there is a conceptual disconnect that severely reduces the effectiveness and usefulness of these methods. Our approach, by contrast, leverages information available at both levels.

Our fault localization method falls into the category of graph theoretical approaches. These approaches use model of a system to describe a set of symptoms which can be observed if a specific fault occurs. The fault localization algorithm then analyzes the model to identify explanations of the observed failures. These techniques [37, 58, 31, 39, 59, 13] are particularly suited for fault localization of distributed systems (e.g. service-based systems), because they allow to represent the complex hierarchical and multi-level structure of these systems.

The graph theoretical approaches can be classified into two categories based on the type of a system model they use.

The first category uses a dependence graph of the system to represent a set of system components and their dependencies [37, 58]. Although dependence graph provides natural representation of the analyzed system, this type of methods is usually limited to modeling only single type of failure per system component.

The second category uses a fault propagation model (FPM) which is a type of causality graph to represent various types of events occurring in the system components and the cause and effect relationships between these events [31, 39]. The FPM thus represents detailed model of the analyzed system that allows the fault localization to analyze system events (i.e. symptoms) rather than the complex multi-state system components as in the dependence graph based approaches. The fault localization approaches based on the FPM provide a set of candidate root-causes explaining the observed symptoms. It have been demonstrated that the analysis of the FPM is an NP hard problem [13, 37].

In order to manage the complexity of the fault localization analysis based on the FPM, the existing approaches typically impose certain limitations on the FPM. The shape of the FPM may be restricted to bipartite graph such as in [14, 39, 59]. Other approaches restrict the number or types of faults included in the FPM, such as [39, 13].

Our fault localization method does not impose any limitations on the size of the FPM or on the number of included symptoms. Instead, the FPM is constructed from dependencies and symptoms occurring in period of time relevant to the analyzed fault. In this way, only relevant services and symptoms are included in the FPM. Moreover, prior to the fault localization, irrelevant root-cause candidates are eliminated from the FPM based on analysis of cause and effect of fault symptoms.

2.3. Distributed Data Harvesting

The problem of transfer of data between nodes with low connectivity in MANETs has been studied extensively. The domains of existing research can be classified into the following three categories.

A Distributed Hash Table (DHT) based techniques are frequently used to facilitate data transfer in MANETs. Number of adaptations of DHT for MANET has been proposed (see Heer et al. [34] and Oliveira et al. [49] for a review and comparison). In general the DHT provides hash-table like distributed lookup service, which uses hashed keys to index and search for network resources such as IP addresses of nodes or files stored on network nodes. The DHT is used in content sharing methods such as in P2P file sharing in MANETs [41, 38, 21]. The DHT is also used in several routing algorithms to enable routing in MANETs as well as to manage real time data streaming in MANETs [42]. Although these techniques facilitate transfer of data in MANETs, they do not address the fundamental problem of limited connectivity between nodes in MANETs and thus are not relevant for transfer of monitoring data.

The second category includes number of data replication techniques proposed for MANETs (see Padmanabhan et al. [51] for a survey). The goal of the data replication is to increase availability of the data by creating and maintaining replicas of databases on multiple nodes in the network. The data are replicated from its originating source nodes closer to the nodes consuming the data. In order to maintain high availability of the data in the replicas, the data have to be regularly updated. However, the process of the data update is subject to the same network limitations as any other data transfer mechanism in the MANET. Therefore, due to the mobility and partitioning of the MANETs, the data replication techniques face challenges in maintaining consistency and availably of the data. The data replication techniques thus make tradeoffs between some of the replication constraints.

Number of techniques focuses on placement and maintenance of replicas based on information provided by or known in advance about the consumers of the data. Karumanchi et al. [36] address problems of network partitioning by creating multiple replicas of data based on consumers advertising about which data and where will be needed. Although this approach provides high data availability, the technique may cause significant network overhead depending on the mobility and size of the network. Our method has to cause minimal network overhead because it is an additional network management tool rather than the primary functionality of the system. Hara [28] designed several techniques for replication of static read only data, and in following work [29, 30] of data with incremental updates. The techniques optimize placement of replicas based on various types of in advance knowledge such as where, which type of data and how frequently will be consumed. Bellavista et al. [11] presented technique which requires similar in advance input, designed for optimal dynamic placement of replicas in MANETs with dense topology. Furthermore, in [10] is presented algorithm for efficient data retrieval from multiple replicas holding partial data. These techniques increase availability of the data in the network, however, do not address problems of the network partitioning. Furthermore, in the fault localization and dependence discovery, the information about where and when the monitoring data will be needed is not known in advance.

Another type of data replication techniques optimizes placement of replicas such to minimize the overall amount of data transferred in network [50, 46, 44, 56]. The techniques minimize the amount of data transferred either by limiting the locations where data can be created [50], or by transferring infrequently large amount of changes in bulk messages [46, 44], or by optimizing the locking of data for read and write operations [56]. The limitation of these techniques is the inability to address the network partitioning and thus they do not provide sufficient data availability.

Another type of data replication techniques focuses on placement and maintenance of replicas in order to address the MANET networks partitioning. These techniques are designed to maintain high data availability on possibly frequently disconnected nodes. Wang et al. [65] identifies patterns in mobility and data use of groups of nodes. Once the nodes are categorized into the groups, the replicas are placed such to increase availability to members of the groups. Huang et al. [35] identifies patterns in mobility of groups of nodes to periodically redistribute replicas of data. Hauspie et al. [32] attempts to predict the network partitioning by measuring quality of links and end-to-end paths and proactively creates replicas. Although these techniques increase availability of data in the partitioned networks, they frequently cause high data overload. Furthermore, these approaches tend to rely on patterns in mobility and data use of the nodes. However, in the fault localization and dependence discovery such patterns do not emerge.

The aforementioned data replication techniques display certain common properties, such as the classification of the network nodes into clients consuming and creating data and servers storing data in databases and maintaining replicas. This classification is not suitable for the monitoring data because in the service-based systems, every node is potentially creating data (i.e. monitoring the locally hosted services) and consuming data from other nodes (i.e. symptoms and dependencies). Furthermore, most of the data replication techniques focus on transaction oriented data access control. These mechanisms increase complexity and data overhead and are not useful in the management of monitoring data because the data are always created on single node and consumed on multiple other nodes, thus removing the need for transaction management altogether.

The third category of data transfer approaches designed for MANETs are epidemic protocols (see Dimakis et al. [20] and Birman et al. [12] for surveys). The epidemic protocols (or gossip protocols) are a category of distributed algorithms which use periodic interactions between nodes to propagate data in the network. Particularly relevant to the problem of data transfer in MANETs is that the protocols do not require reliable communication links. The peers in interactions are usually chosen randomly and there is a certain delay between the start of the data transfer from the source node and the reception of the data at the target nodes.

In context of the MANETs, the epidemic protocols are frequently used for in-network data processing such as in distributed signal processing [67, 55]. In these techniques, the epidemic protocols are used to decrease the overall network overhead by creating overlay network layer which allows computation of an intermediate values and passing on aggregated data. Other approaches were proposed for data replication purposes [44] with limitations listed above.

Our method makes use of the ability of the epidemic protocols to create overlay network to pass on data on the unreliable links similarly as the distributed signal processing approaches. However, instead of to transfer only aggregates, our method passes on incremental changes in similar fashion as some of the data synchronization approaches.

3. Dependence Discovery

In this chapter we present our dependence discovery method. We begin by describing the two types of dependencies in which we are interested. We then describe how we discover dependencies, represent dependencies, and construct the representation.

3.1. Introduction

In service-based systems, a *dependence* is a relation between services defined by the message flow induced by a client request. (As an edge case, a dependence is also the relation between a client and a service. Without loss of generality, we mainly focus here on relations among services.) When a dependence relation exists between two services S1 and S2, one service is considered the *source* and the other the *target*. In general, sources issue requests on targets, thus defining a directionality to the dependence.

We are concerned with two types of dependencies over a given set of services: *inter-dependencies* and *intra-dependencies*. An inter-dependence is the basic dependence relation that exists between the requester of a service and the receiver of that request. Figure 3.1 illustrates logial set of inter-dependencies in a system, where the arrows indicate the directionality of the dependencies, from sources to targets. For instance, service S3 is directly dependent upon services S9, S11 and S20, and indirectly dependent upon services S10, S12, S15, S21, S22, S23, and S25. Each inter-dependence in which a particular service is engaged can be classified as either *incoming* or *outgoing*. Service S9 has an incoming inter-dependencies with S2 and S3 and outgoing inter-dependencies with S10 and S12. The figure further exposes highlighted subset of services and dependencies involved in single conversation of client C3. In this conversation service S3 uses services S9 and S20 to complete the request while service S11 is used by S3 in processing of other coversations.

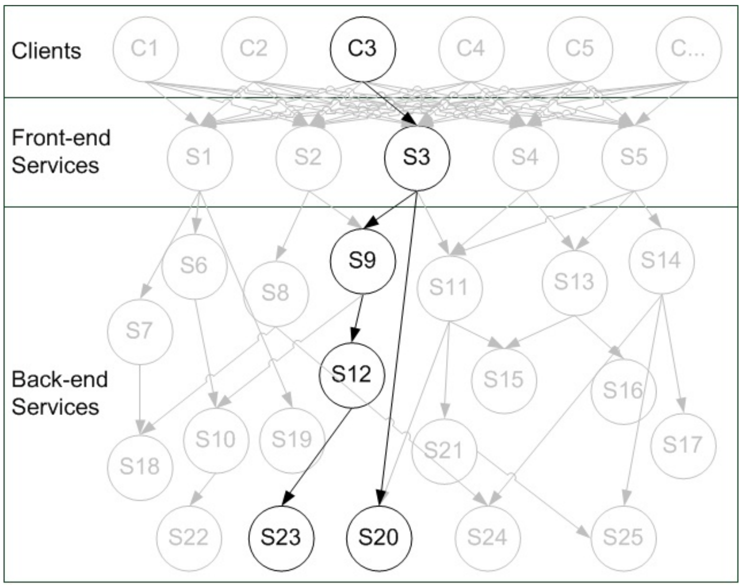


Figure 3.1.: Inter-dependencies of service-based system in MANET, with highlighted single conversation of client C3

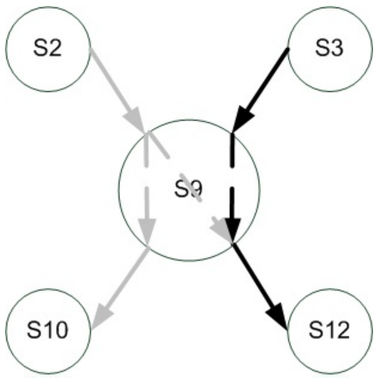


Figure 3.2.: Intra-dependencies of service S13

An intra-dependence is a more complex relation between services that relates an incoming inter-dependence to an outgoing inter-dependence. In this sense, intra-dependencies reflect more detailed insight into the nature of the dependencies between services than do the basic inter-dependencies. This is illustrated in Figure 3.2, which shows the dependencies among S2, S3, S9, S10, and S12 resulting from the four given inter-dependencies (solid arrows) and the three given intra-dependencies (dashed arrows). It is instructive to compare the information gained from Figure 3.2 to that available in Figure 3.1. We can see that S2 is (indirectly) dependent upon S10 and S12, while S3 is only (indirectly) dependent upon S12.

This is not evident from Figure 3.1.

3.2. Discovering Dependencies

As mentioned above, dependencies arise from the flow of messages among services. To discover dependencies, we must therefore track these flows. Because our aim is to be minimally intrusive, we restrict ourselves to observing the message traffic (i.e., messages that contain service requests and responses) as it occurs. Our method makes use of *monitors* deployed within the network to observe messages and record information about the flows. A convenient place to deploy a monitor is within a service's container. The monitor is then easily aware of the associated service's identity, as well as being provided a context in which to execute.

The main advantage of an approach based on monitors is that it allows us to discover dependencies instantaneously and precisely, with minimal delays between dependence occurrence, detection, and the availability of the dependence information. Moreover, we can do so without having to modify the services themselves. Monitors can also minimize data storage and communication requirements, since they can actively aggregate and summarize the information. Thus, our approach can be thought of as a process for collecting evidence of dependencies, which is in sharp contrast to methods that require storage and transfer of large amounts of data for later statistical analysis.

In basing our method on monitors, we assume that they can be deployed into service containers and are able interpret the messages they observe. Important system requirement is correctly synchronized system time on nodes with monitoring agents, which is required for correct marking and matching of detected dependencies.

Inter-Dependence Discovery Pairs of source and target services that induce inter-dependencies can be identified from the flow of messages exchanged between the services. In a service-based system, services are uniquely identified by application-specific identifiers, such as the URIs of the Web Services framework. Although the specific type of information provided within messages differs with the service platform and standard used, request messages always contain identifiers for the requested services.

Since a monitor is aware of the identity of the service with which it shares a container, it can record *outgoing inter-dependencies* simply by extracting the identifiers of target services from any outgoing request messages originating at the service. The target service identifier is an essential field present in all request messages, such as plain HTTP or SOAP requests. The inter-dependencies are therefore easily discoverable in all existing service invocation protocols such as SOAP, REST or even from plain HTTP requests without requiring any modifications to existing systems.

Intra-Dependence Discovery Intra-dependence discovery requires knowledge of both outgoing as well as incoming inter-dependencies. However, discovering *incoming inter-dependencies* is a bit more involved, as it requires request messages to contain the unique identifier of the requesting service. This is satisfied by most service standards (e.g., the WS-Addressing standard provides the fields `wsa:To` and `wsa:From`). Alternatively, when field containing identifier of the requesting service is not present in the request messages, the monitor can insert such a field into a request message's header when the message is being sent from the requesting service. With source and target identifiers present in messages, monitors can detect all incoming and outgoing inter-dependencies.

To expose the correspondence between the incoming and outgoing inter-dependencies of a service, we rely on the presence of *conversation identifiers* within messages. In service-based systems, a conversation is the set of all messages exchanged during the processing of a single client request. Typically, the conversation is identified across the messages using tags inserted into message headers; the tags contain a unique conversation identifier. Of course, conversation identifiers must be implemented by service programmers. Fortunately, this is a relatively routine task, as there are several existing standards for doing so, including WS-Addressing,[1] WS-SecureConversation,[2] and WS-Coordination,[3] that are increasingly used in current-day applications. Discovering an intra-dependence then reduces to having a monitor relate incoming and outgoing messages using the conversation identifiers appearing in both.

Alternatives to our *dynamic* approach of the intra-dependence discovery exist. For example, in early stages of our work we have also explored possibility to use static code analysis [25]. In a stable environment of a fixed networks, the information about the intra-dependencies can be "statically" hardcoded into the source code of a service or the service may have associated stable configuration files. The static code analysis may discover dependencies by examining either source code, compiled output and/or configuration of the service. However, in an environment with a dynamic topology, the information about target services has to be dynamically obtained from a service discovery mechanism and thus the service code or the configuration files cannot contain enough information to identify

[1]`http://www.w3.org/2002/ws/addr/`
[2]`http://www.ibm.com/developerworks/library/specification/ws-secon/`
[3]`http://www.ibm.com/developerworks/library/specification/ws-tx/`

the target services.

3.3. Storage of Dependence Data

Extracting the source and target fields of messages in the service container is a simple read-only operation causing minimal computational overhead. However, in the resource constrained environment of MANETs, crucial aspect of the monitor is the data storage requirements.

A monitor will maintain the history of the dependencies it has seen for some bounded period of time, and in some storage data structure. The dependence data are stored within the data structure such that only a limited history is maintained. The data structure can be designed to either store individual timestamps of the dependence occurrences or divide the dependence occurrences into time slots.

Timestamp based data structure When individual timestamps of dependence occurrences are stored, the size of the stored data is dependent on workload generated in the system. Thus the size of the storage might be significant in high workload environments. Yet, this mechanism is highly precise because all of the dependence data are maintained. Therefore, this option is suitable when high data precision is desired and when higher amount of stored data can be tolerated such as for system testing purposes. Conversely, in standard production environment the occurrences are stored in space efficient data structure.

Time slot based data structure In production environment, a monitor will maintain the history of the dependencies divided into time slots. The dependence data are stored with a sliding expiration window such that only a limited history is maintained, using a data structure representing sliding time slots. Entries for each time slot maintain Boolean data about whether or not a given dependence occurred within that time slot. Each dependence is associated with a set of those time slots, such that when the monitor detects the occurrence of a dependence, it signifies this by setting a 1-bit flag in the corresponding time slot. It also records identifying information about the source and target of the dependence. Figure 3.3 illustrates the setting of the bit flags in corresponding time slots in the represented time period.

Of course, the size of the time slot affects the precision of the data maintained. For example, a slot size of 0.1 seconds will provide up to 10 times more data compared to a slot size of 1 second, but will require 10 times more space to store those data. Beyond the

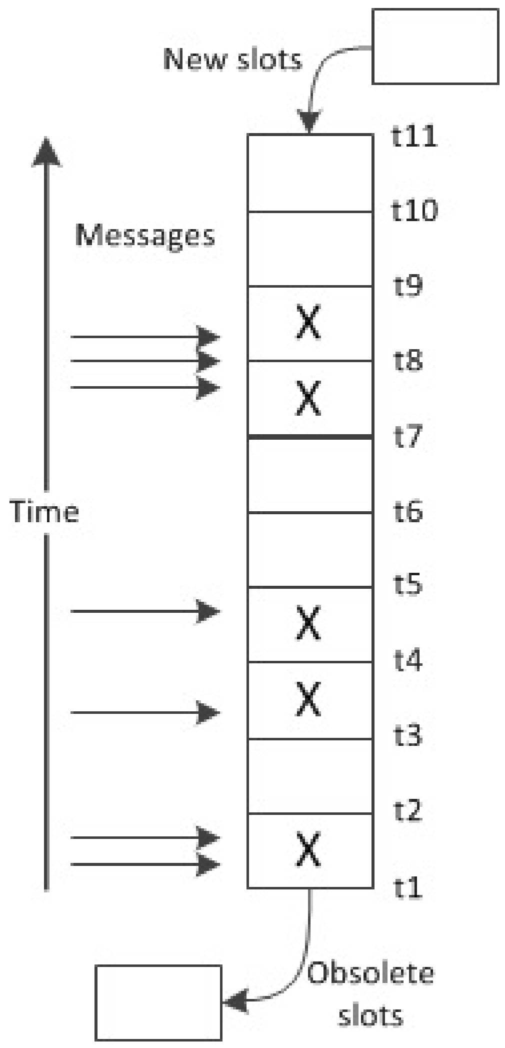


Figure 3.3.: Setting of the bit flags in corresponding time slots maintained by monitor

slot size, the size of the whole history can be controlled through the pruning of expired time slots.

The length which each time slot represents can be configured according to a desired level of resolution. When combined with the time period and the size of each dependence entry, the data storage needs for each monitor is determined. The estimated number of bytes of storage space S per monitor is as follows:

$$S = \frac{T_h}{T_s} \times \frac{(N_{inter} + N_{intra})}{8} + (B_{inter}N_{inter} + B_{intra}N_{intra})$$

where T_h is the length in seconds of the time period, T_s is the length in seconds of a time slot, N_{inter} and N_{intra} are the average number of inter-dependencies (both incoming and outgoing) and intra-dependencies recorded by an monitor during the time period, and B_{inter} and B_{intra} are the maximum sizes of the identifiers of inter- and intra-dependencies. We provide example of estimation of the data storage costs in the evaluation section.

The aggregation of observed service interactions into dependencies can cause the monitors to "forget" some of the details necessary to reconstruct accurate dependence information. This is the case when internal behaviors of a service that are not visible to the monitor can generate seemingly identical interactions in aggregate. As an example, consider the incoming inter-dependence of S2 on S9, and the outgoing inter-dependencies on S10 and S12, shown in Figure 3.2. The incoming inter-dependence could in fact be the result of aggregating two separate conversations between S2 and S9, where the first resulted in a message to S10 and the second resulted in a message to S12. The aggregated dependence information held by the monitor will not, in our current approach, record the true dependencies regarding these individual conversations.

The tradeoff between storing timestamps or time slots is a loss of precision of the dependence data caused by the aggregation of the dependence occurrences into single Boolean value per time slot. However, unlike storing occurrence timestamps where the size of the stored data depends on number of stored occurrences and thus is sensitive to system workload, the time slot algorithm requires fixed amount of memory to store the data regardless of the system workload. Thus for example to store data of single dependence with 10 occurrences within one minute; the timestamp based storage will require 80 bytes of data (with 8 bytes required to store one timestamp of millisecond precision in Java) and the time slot based algorithm will require 75 bytes of data with 0.1 second resolution of the time slots. However, with 10x higher workload of 100 dependence occurrences over same period of time the data requirements will increase to 800 bytes for the timestamps and stay same at 75 bytes for the time slots.

3.4. Dependence Graph

A dependence graph (DG) is a directed acyclic graph constructed from a set of nodes representing services and a set of edges representing direct inter-dependencies. The direction of an edge represents the direction of the inter-dependence, from source to target. Each node can be annotated with intra-dependence information, conceptually adding directed edges between the incoming and outgoing inter-dependencies of the service.

The DG maintains information concerning a specific *time window*, reflecting only the dependence information collected by (or perhaps available from) monitors during that period. The time window is a property of the interaction between the application behavior, the network behavior, and the information accessible to monitors. The size of the time window has many effects on the results of analysis. For example, a small time window serves to reduce the size of the DG, but some critical service interactions might be missed. A large time window provides a more complete record of dependencies, but might include stale or irrelevant interactions (e.g., those belong to conversations other than the target conversation for the analysis).

Conceptually, a DG could be used to represent the full set of dependencies of an entire application system. In practice, many analysis techniques only require a subgraph of the full dependence graph related to a specific node or subset of nodes. For example, a failure impact analysis might examine only the nodes that can reach (i.e., are dependent upon) a given node, and a fault localization analysis might examine only the nodes that are reachable from a given node.

3.5. Dependence Graph Construction

Our method uses a set of distributed monitors that provide information to a dependence discovery element, as illustrated in Figure 3.4. The intent of this architecture is to minimize resource utilization, while still providing timely data. The monitors perform continuous dependence discovery and maintain aggregated dependence data (unless in exceptional cases timestamps of dependence occurrences are maintained as described in 3.3).

The availability of the dependence data during the construction of the DG is subject to reachability of nodes hosting the monitors from nodes conducting the dependence discovery and data harvesting. Typically in MANETs, the reachability between various nodes is limited due to network spatial-temporal partitioning as well as due to variable quality of the network links. Thus the availability of the dependence data must be considered as an important design aspect of the dependence discovery method.

We have designed two alternative data harvesting algorithms each suitable for different scenario of reachability between network nodes. In this chapter we describe an approach, which is based on the assumption of full reachability of the network nodes and thus on full availability of the dependence data on demand from monitors. In Chapter 5 we provide distributed approach of data transmission suitable for scenarios in which the reachability of nodes is limited and thus the availability of the dependence data on demand is not guaranteed.

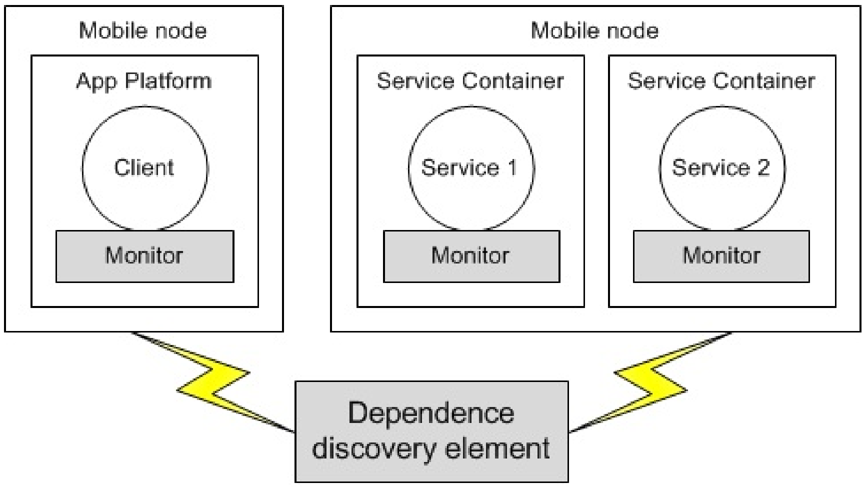


Figure 3.4.: Architecture of dependence discovery

The discovery element will construct a DG on demand, querying the relevant monitors to harvest their local dependence data for the time window of interest. The harvesting algorithm may either attempt to transmit the requested data on demand at the time when the dependence graph is being constructed or it may use some distributed algorithm to transmit the data before the dependence graph construction has started. The construction algorithm is designed to incrementally construct the DG—typically a subgraph of the full application dependence graph—by visiting only the monitors considered relevant based on the data seen to that point. In this way, the amount of data transmitted over the network can be significantly reduced compared to existing methods. Of course, the most common case of DG construction is for a particular client, revealing the services upon which that client depends directly or indirectly. Conceptually, the data are harvested by a walk rooted at the monitor associated with the client.

For certain analyses, we may additionally want to limit the dependence discovery to a specific conversation, something particularly useful in fault localization. This can be performed by adjusting the time window of the DG.

In Figure 3.5 is shown an activity diagram conceptually representing the dependence discovery algorithm. In the first step, the algorithm adds root dependencies into the DG. The root dependencies might be limited to a single root dependence. This is useful when discovering DG of particular conversation such as in the fault localization. Alternatively, the root dependencies include all of the root dependencies occurring within the time window.

This is useful when discovering set of all services involved in requests in particular time period such as in composition analysis. In the following steps, recursively, the algorithm sends request for outgoing dependencies of each of the services added into the DG.

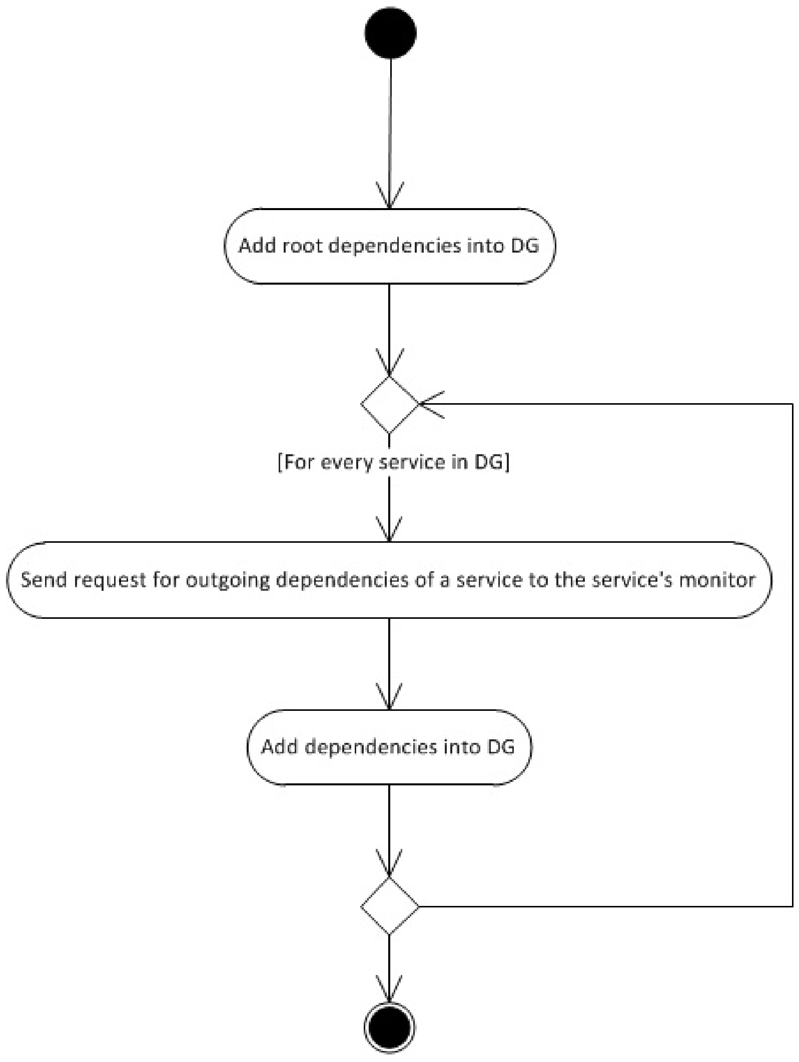

Add root dependencies into DG
[For every service in DG]
Send request for outgoing dependencies of a service to the service's monitor
Add dependencies into DG

Figure 3.5.: Activity diagram of dependence discovery algorithm

Finally, in Appendix A.1 is shown an example of the dependence discovery algorithm implemented in Java. The algorithm has input parameters of identifier of a root node (i.e. client) and timestamps of beginning and end of the time window. The algorithm uses function which queries monitors in network for outgoing dependencies of services and constructs DG in breadth-first search sequence. The algorithm initially creates new object representing the DG with single root node representing the client's node. Subsequently, in loop for each node in the DG is invoked the method getDependenciesForNode. Based on the node parameter, the method either queries the local monitor for the client's dependencies or sends request to monitor hosted on another network node. For every dependence added into the DG, a new node representing the dependence's target service is created in the DG. For each node in the DG is invoked the method getDependenciesForNode to get its outgoing dependencies within the time window.

The dependence discovery element can be hosted on any node of the network. Moreover, since the dependence discovery element is a small software component, it can be hosted on multiple nodes of the network. For example, every node which hosts some client application is also a good candidate for hosting the dependence discovery element, either as a component of fault localization or as a component of other analytical methods.

3.6. Summary

In this chapter we have presented our dependence discovery method. We have described the two types of dependencies the method discovers, the techniques used to discover the dependencies, the dependence graph the method builds, and the architecture of the method.

In Section 6.1.5 we describe the experimental tools we used to evaluate the method, followed by description of the implementation of the method in Java EE provided in Section 6.2.2.

The accuracy of the dependence graph construction mostly depends on the time window and workload. With a short time window and a small workload (i.e., infrequent interactions among services), we may not observe some critical dependencies. With a long time window and a large workload, we may include obsolete and irrelevant interactions (e.g., dependencies belonging to conversations that are not of our interest).

In Section 7.1, we investigate the impact of these and other factors on the accuracy of the DG obtained by our method. We then suggest ways to select parameter values to optimize the discovery process.

4. Fault Localization

In this chapter we present our fault localization method. We begin by describing how faults propagate within service-based systems. We then describe the architecture and assumptions of our method, followed by description of the fault localization technique and the two types of ranking algorithms.

4.1. Introduction

In service-based systems, a *fault* is an exceptional condition occurring in a software (i.e. service) or a hardware (i.e. network link or node) components. A fault which occurs in processing of a conversation propagates in a series of *failures* of services back to the client which initiated the conversation. The client receives notification of failed conversation instead of the requested functionality. Typically, failure does not carry any information about its cause or source. Thus, upon receiving the failure, the client is left without any explanation about the cause of the failure.

The failures received by the clients of the service-based systems can be categorized into *exceptions* which represent explicit notifications of failures received in response messages and *timeouts* which represent implicit notifications of failures caused by response messages not arrived within a specified period of time.

The goal of our fault localization method is to find the root cause (i.e. the root cause fault) of a failure in a conversation initiated by a client of a service-based system. The conversation failure received by the client is hence an entry point of our fault localization method. The method makes use of both service- and network-level information and traces the failure back to its source.

When fault or failure occurs in a system component it manifests as a *symptom* and thus can be intercepted and recorded into a system log. To trace the propagation of failures through the system components (i.e. the services, nodes, links and others), we therefore track these symptoms. In order to be minimally intrusive, we restrict ourselves to extracting the records of the symptoms from the system logs.

The particular type of the root cause (i.e. fail-stop, Byzantine, or any other) is not significant from the point of view of the analysis. As long as the conversation failed and the root cause fault caused propagation of the faults through services and issuance of the associated symptoms, the method can be used to trace the source. However, the method is not designed to address localization of faults which do not cause the conversation to fail (e.g. Byzantine faults causing incorrect results).

4.1.1. Terminology

Before describing our technique in detail, we first introduce the terminology used throughout the description.

Fault: An exceptional condition occurring in a software or hardware component. A fault may or may not cause a failure in another component.

Failure: An external and visible manifestation of a fault as a deviation from the expected behavior of a component.

Fault propagation: A failure in one component may cause failures in dependent components that are otherwise fault free. Thus, a fault may propagate via a cascade of failures.

Root cause: The apparent origin of a fault propagation. We use the more specific terms *root-cause fault* and *root-cause failure* interchangeably, where the latter is the first visible (and, hence, observable) manifestation of the former.

Transitive failure: A failure occurring in dependent components due to propagation of the fault from its root cause to the client application through system components. The failure experienced by client is a case of transitive failure at the end of the fault propagation.

Symptom: An observable manifestation of a fault or failure. Symptoms are typically recorded in a system log and include ancillary descriptive information. We consider here following two specific kinds of symptoms:

- ***Exceptions*** are explicit notifications of failures carried in response messages sent from target services back to source services. They are initiated by software components of services which failed to accomplish some task e.g. were unable to send request to another service. The cause of the exception can be any failure, whether originating in the network, hardware or software components.

  Note: The exception are propagating through software components (by the standard throw-try-catch mechanism) and through response messages (as a specific type of response message defined and managed by the platform, standard and technologies used in the system).

- ***Timeouts*** are implicit notifications of failures caused when an expected response message has not arrived within a specified period of time. They are caused by either the network failing to transfer responses back to source of requests or by software or hardware components failing to send response message.

4.2. System Architecture and Assumptions

The architecture of the fault localization method is built upon the architecture of the dependence discovery method presented in Section 3.5. The fault localization method uses a set of distributed monitors, as illustrated in Figure 4.1. The monitors are deployed in service components (e.g., in a service container) and are responsible for extracting the symptoms of service failures from system logs. The monitors provide the symptoms, on demand, to a fault localization element (e.g., located at an operator's station) that runs the fault localization method.

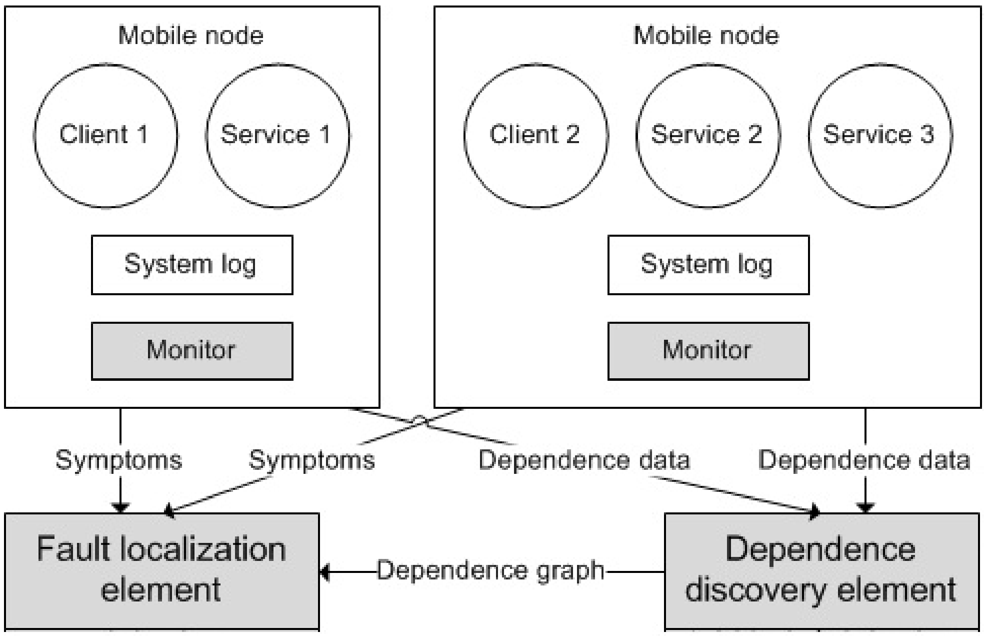


Figure 4.1.: Architecture of fault localization

An essential input to our method is a *dependence graph* (DG) that captures the run-time dependencies among services. In the previous Chapter 3, we have presented a method to obtain probabilistically accurate DGs in a MANET-hosted, service-based system.

A DG can be used to represent the full set of dependence relations in the system, or can be restricted to a subset of those relations. For our purposes, we restrict the DG to the dependencies rooted at a particular client and for a particular time period. Figure 4.2 shows a simple example of a DG rooted at a particular client.

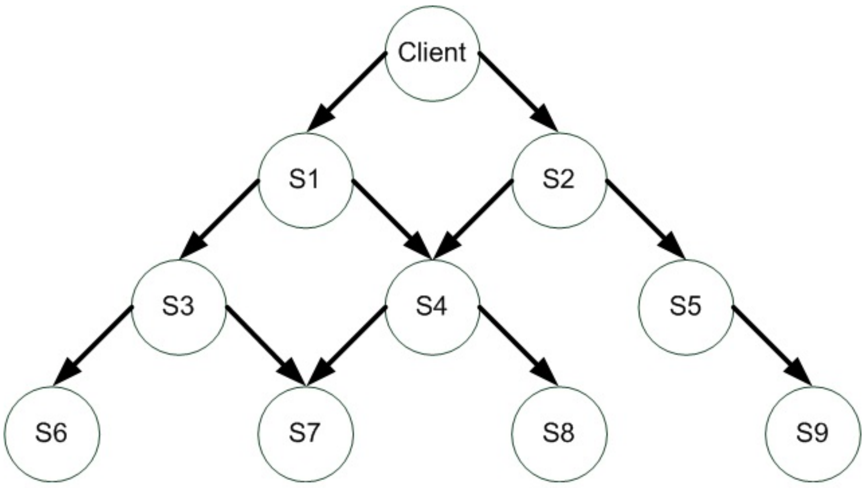


Figure 4.2.: Dependence graph rooted at a client

The dependence discovery method is designed to take account of the peculiarities of that environment, specifically the dynamic nature of service-level binding and host-level mobility, combined with the general lack of resources for tracking dependencies. The method works by building (i.e., “discovering”) the DG for a client on demand, which is an approach that is particularly well suited to fault localization because it allows us to obtain the DG for the client that has reported a failed conversation. Of course, due to the difficult operational environment, the resulting DG can be expected to contain some false dependencies, as well as miss some true dependencies, although the discovery method is designed to minimize the occurrence of such inaccuracies. We must therefore take account of this in the design of the fault localization method. Later in our experimental results, we evaluate the impact of the accuracy of the DG’s on the performance of our fault localization method.

Besides the availability of dependence information, we make the following assumptions.

1. Each service records all failure symptoms in a local log, and the local monitor has access to the log.

2. The log entries include information such as a time stamp, an identifier for the relevant service instance, and the type of symptom.

3. Time stamps are globally consistent due to synchronized clocks. Clock synchronization in MANETs is a well-researched topic, with techniques available to achieve precision of tens or even single microseconds [68]. The smallest time period we use to distinguish time stamps is on the order of several milliseconds, well within this precision. The cost of the time synchronization required may be none or very small. The time synchronization functionality is an integral part of a system management and is usually required by a low level network protocols and thus in place by default. In case when time synchronization must be introduced due to our method, the cost will be negligible. The overhead imposed on the network by the time synchronization is related to the precision required. Since our method requires low precision, the overhead will be very small.

4. Finally, there is a global timeout parameter shared by all clients and services. The timeout parameter defines the maximum waiting period for responses to requests. This is easily achieved in systems that use a common application platform, such as .NET or J2EE, which have a predefined default value. In systems that use a mix of platforms, explicit enforcement would be required.

4.3. Fault Localization Technique

The previous section introduces the general architecture we envision for recording and gathering the information needed for fault localization. Assuming this information is available to the fault localization element shown in Figure 4.1, our method for analyzing the information shown in Figure 4.3 consists of the following high-level steps:

1. *Mapping*: A *fault propagation pattern* (describes how faults can propagate as failures from one component in a system to another, Section 4.3.1) is mapped onto the dependence graph associated with a failed conversation, resulting in a *fault propagation model* (represents the causality relations between failure events in a system, Section 4.3.2) that represents how specific faults might propagate through the individual services involved in the conversation.

2. *Reduction*: The fault propagation model is combined with the occurrence times of relevant symptoms. This assignment allows the method to reconstruct the possible

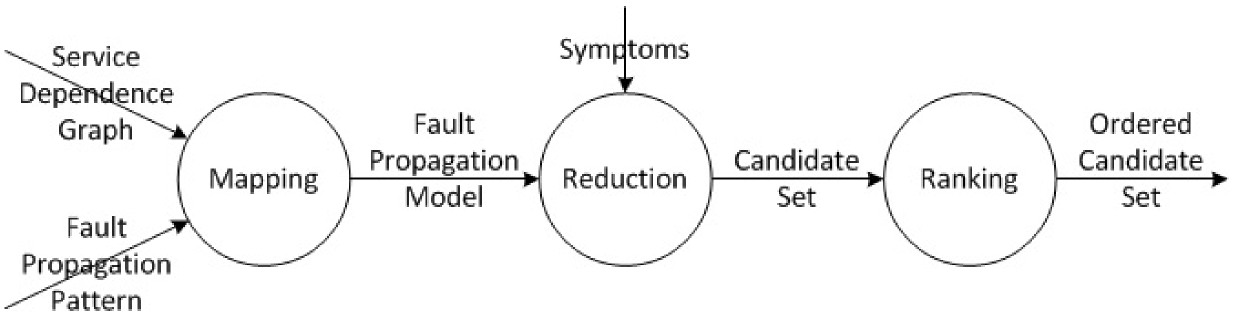


Figure 4.3.: Fault localization method

propagation paths for the fault, both in time and in space, and thus to form a set of plausible *candidates* for the root cause of the failure.

3. *Ranking*: A *ranking* (Section 4.3.3) is applied to the elements of the candidate set based on their likelihood of being the root cause of the failure. We have devised two alternative ranking algorithms based on established fault localization techniques. The first is a timing-based approach that ranks hypotheses based on the time difference between possible root causes and the client's fault. The second is a probabilistic method that uses a Bayesian network to infer independent probabilities of individual root-cause hypotheses. We present a comparative evaluation of the effectiveness of these two alternatives in a later section.

4.3.1. Fault propagation pattern (FPP)

We describe how faults can propagate as failures from one component in a system to another by defining a *fault propagation pattern* (FPP). More precisely, an FPP is a recursive description of how different kinds of failures can cause further failures, where the first failure in the propagation chain is the failure manifesting the root-cause fault. In service-based systems, the propagation flows upstream from target services back to source services in the service dependence structure.

To enable a fine-grained analysis using propagation patterns, we classify the observable failure symptoms—exceptions and timeouts—into failure *modes*.

A propagated *exception* (Figure 4.4a) can be generated as a result of one of the following modes: (1) a send failure (SENDF), which represents a service that is unable to send a message to another service due to a network-level fault; (2) a software failure (SF), which represents any internal software or data fault that throws an exception; and (3) an exception

(EX), which is itself generated by a service in response to a failure in a downstream service. Notice that the first two are (manifestations of) actual root causes, while the third is the recursive behavior that leads to propagation .

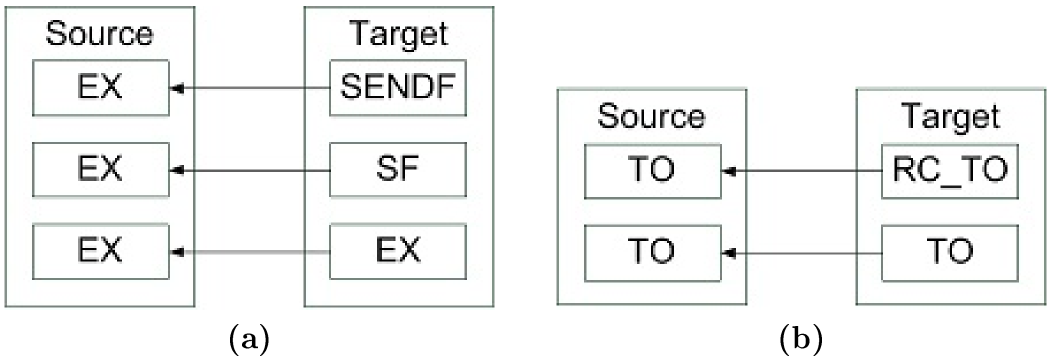


Figure 4.4.: Fault propagation pattern refined into modes for exception failures (a) and timeout failures (b) in service-based systems. Faults propagate upstream from targets back to sources in the service dependence structure

A propagated *timeout* can be caused only by some other timeout. This is because the timeout event is an implicit symptom of a fault whose real cause is not directly observable and, hence, not itself recordable as a symptom. Examples include a network host that moves out of communication range or the physical failure of a host. We identify two timeout modes (Figure 4.4b) that can cause further upstream timeout failures: (1) a root-cause timeout mode (RC_TO), which results implicitly from a network-level fault, and (2) a transitive timeout mode (TO), which results from a timeout occurring in some downstream service.

A subtle and counter-intuitive point: A timeout failure is consistently witnessed by a client *before* the root-cause RC_TO failure occurs. This is because the client starts its response timer when it initiates the conversation, and all downstream timers in the conversation are also set to this same default value, as mentioned in Section 1. Thus, the client's timer would actually be the *first* to expire in the conversation.

4.3.2. Fault propagation model (FPM)

Our technique makes use of a graph, called a *fault propagation model* (FPM), representing the causality relations between failure events in a system. An FPM is the result of mapping the fault propagation pattern (FPP) described above onto the service dependence graph (DG) of a given client. It is a rooted reverse-directed acyclic graph whose nodes and edges correspond to the client, services, and dependencies of the DG. More precisely, the graph contains a single node at its root to represent the failure mode witnessed by the client, and a set of nodes at its leaves to represent the failure modes of the candidate root causes. The internal nodes represent transitive failure modes. The edges signify possible failure propagation paths, which follow the reverse dependence edges in the DG. Figure 4.5a shows an example constructed by mapping the FPP of exception of Figure 4.4a onto the DG of Figure 4.2.

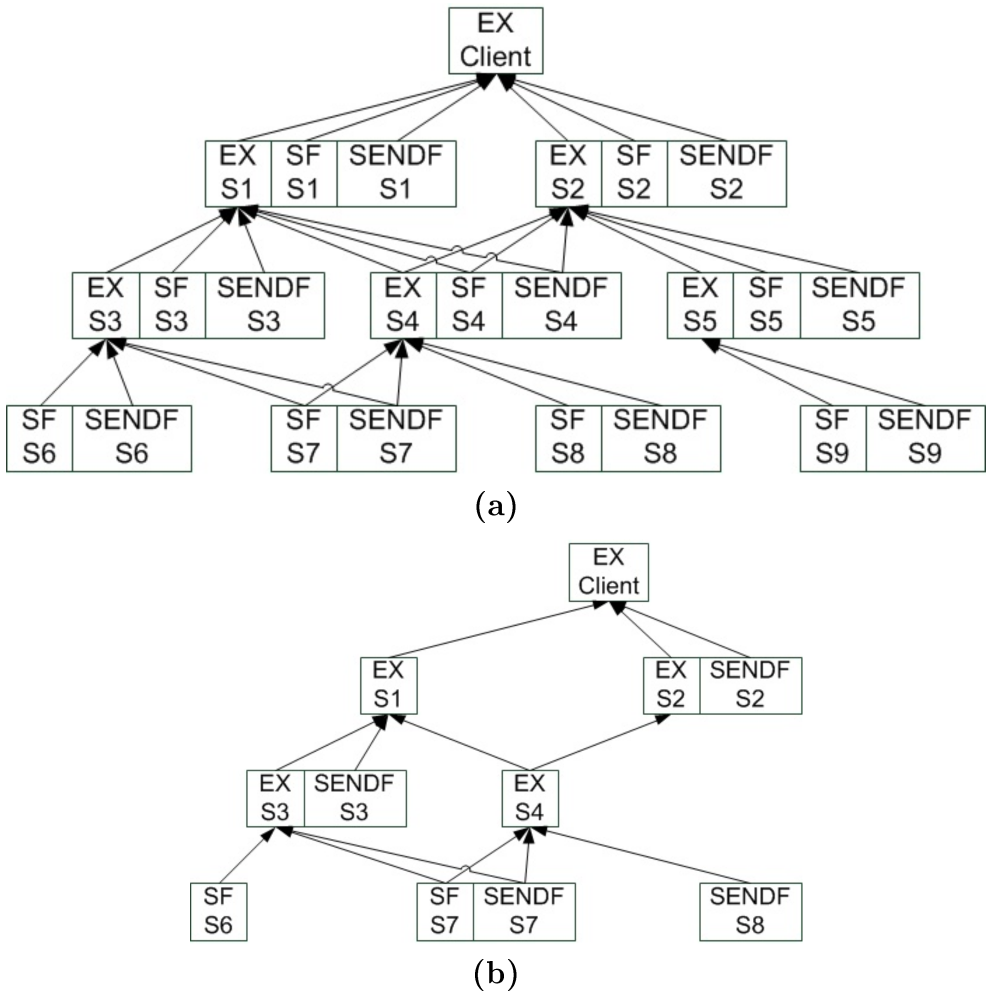


Figure 4.5.: Fault propagation model constructed from exception modes of Figure 4.4a mapped onto dependence graph of Figure 4.2. The full FPM (a) is reduced (b) by considering actual symptoms recorded in a given time window

Faults tend to be temporary in MANETs and can affect the clients only for a limited period of time. In order to capture this temporal aspect of the problem, an FPM typically maintains information concerning a specific *time window*, reflecting only the failure symptoms collected by monitors during that period. The time window is a property of the interaction between the application behavior, the network behavior, and the information accessible to monitors. The selection of the time window size is critical in the construction of the FPM. For example, a small time window serves to reduce the size of the FPM, but some critical service interactions and symptoms might be missed. A large time window provides a more complete record of dependencies and symptoms, but might include stale or irrelevant interactions and symptoms (i.e, those belonging to conversations other than the conversation of concern to the analysis). In principle, one needs to consider how long it would take all potential root-cause faults to propagate to a client. Ideally, the time period should be long enough to cover the longest such propagation, but no longer, so as not to include considerable noise in the data. We examine this issue carefully in the context of our experimental evaluation (Section 7.2.1).

Operationally, upon receiving a failure report from a client, the fault localization element builds the FPM with respect to the reported failure. To do so, the dependence discovery element is queried for the DG associated with the failed conversation. The DG is then combined with the FPP and transformed into an FPM for the specific kind of failure, according to the following steps:

1. Every leaf service of the DG is transformed into the root-cause failure modes of the FPP; every intermediate node of the DG is transformed into root-cause and transitive failure modes; and the client node is transformed into a single transitive mode representing the client failure.

2. The edges representing cause-and-effect relations between modes are assigned based on the dependencies in the DG such that every mode of a target service is connected to the respective mode of a source service. The resulting FPM represents all possible root-cause modes with respective propagation paths, given the structure of the DG.

3. The fault localization element contacts the monitors of the services included in the DG/FPM in order to harvest locally aggregated symptom data for the time window of interest. The aggregation applied depends on the kind of failure mode associated with the recorded symptom: For exceptions it is the latest in the time window, while for timeouts it is the earliest. (The justification for these aggregations is given in Section 4.3.3.) The data are then assigned to the relevant modes of the FPM.

4. The FPM is *reduced* to a set of candidate root-cause failure modes based on the actual symptoms recorded within the given time window. To do so, any modes (whether transitive or root cause) without associated symptoms are removed from the FPM, as they (probabilistically) could not have either caused or propagated to the client. The FPM is then further reduced to the set of candidate root-cause modes, which is connected to the client failure mode either directly or by propagation paths through the remaining transitive modes. The reduction repeatedly removes all modes which do not have following (parent) transitive mode as well as it removes all transitive modes which do not have any preceding (child) mode.

Appendices A.2 and A.3 show example pseudocode of the construction of the FPM and of the reduction of the FPM algorithms respectively.

The resulting FPM contains the set of candidate root-cause failure modes that are reachable to the client directly or through transitive modes. The FPM also contains the set of transitive modes that are on propagation paths between at least one root-cause mode and the client. For example, the FPM of Figure 4.5a can be reduced to the FPM of Figure 4.5b if we assume the following symptoms are recorded in the given time window: services S6 and S7 experienced software failures (SF); services S2, S3, S7, and S8 experienced message send failures (SENDF); and services S1, S2, S3, and S4 received exception messages (EX). The client, which presumably triggered the analysis, also received an exception message (EX).

The harvesting of the symptom data used in FPM is closely related to harvesting of the dependence data introduced in Section 3.5 because the symptom data are harvested from all nodes of the analyzed DG. Hence the symptom data harvesting mechanism is designed as an extension of the dependence data harvesting mechanism such that the symptom data are transmitted along with the dependence data. In this way, the amount of data transmitted over the network is reduced to minimum and availability of the symptom data matches the analyzed DG.

4.3.3. Ranking candidate faults

Given an FPM that consists of possible root-cause failure modes, the next step is to rank the candidate root-cause faults in the order of their likelihood of being the actual root cause of the client failure. We consider two approaches to ranking the candidates: one based on *occurrence time* and the other based on *Bayesian probabilities*. We compare the two approaches in our experimental evaluation (Section 7.2.2).

Timing-based ranking

The timing-based ranking approach follows from the following observation: Although conversations might last a long time, failures will propagate relatively quickly from root causes to clients. For instance, intermediate services typically generate exception responses immediately upon receiving an exception response from a downstream service in the conversation, ultimately causing the client to see the exception within a relatively short time after the root cause fault has occurred. Similarly, timeout events in the services involved in a conversation tend to occur within a relatively short period. This is due to the fact that services set their timeout timers when they send service requests, and the forward propagation of those requests through the system is typically much quicker than the time it takes for the conversation as a whole to complete.

Based on this observation, our timing-based algorithm ranks the candidate root-cause failures appearing in the reduced FPM in increasing order (from shortest down to longest) in terms of the difference between the time stamp of the client failure symptom and the time of occurrence of the candidate, which is itself determined by the time stamp of:

- the *latest* symptom of the failure, in the case of an exception, and
- the *earliest* symptom of the failure, in the case of a timeout.

This aggregation ensures that the ranking favors the candidate that occurs closest in time to the failure witnessed by the client.

Appendix A.4 provides example pseudocode of the Timing-based ranking algorithm.

Bayesian-based ranking

Our second algorithm assigns the ranks to the candidates in the order of *probability* that the candidate is the root cause. The probabilities are inferred on the Bayesian network (BNet) constructed from the FPM. We use a multi-level BNet model, where each node of the graph is a binary valued random variable that describes the state of a single FPM mode.

The BNet is an isomorphic transformation of the reduced FPM. However, the direction of the BNet edges is reversed from FPM edges because in the BNet we measure the probability of each candidate node to be a root cause, given the evidence that the client witnessed a failure. Thus, we measure the probability of propagation in reverse order.

For the failure propagation probabilities in the BNet, we use the noisy-OR gate distribution model of the conditional probability distribution (CPD), due to its computational and space efficiency [33]. The CPDs are calculated based on two assumptions concerning fault propagation.

- First, given that the reduced FPM only provides the relevant subset of candidate root causes, the probability of the antecedent mode to propagate to a dependent mode is equal for all transitions.
- Second, the probability of a candidate to be the root cause decreases with the increasing number of transitions through which the fault propagates to the client.

Therefore, we calculate the CPD of a node in the BNet such that the conditional probability of a parent node in CPD given the occurrence of the mode in a child node (i.e., the probability of propagation from parent to child node) is equal to the proportional fraction that the parent represents in the set of all parents of the particular child node in the FPM. For example, a child node with two parents has a CPD of 50% propagation probability of/from each parent. Furthermore, in order to reflect the decreasing effect of each transition on the probability of the mode to be the root cause, the conditional probability of each parent is modified with the constant 0.99. This constant is sufficient to distinguish between the candidate modes at different levels of the FPM, yet small enough not to interfere with the parent probabilities.

For the inference of the posterior probability of the root cause modes, we use the junction-tree algorithm [61, 64], which is an exact inference algorithm suitable for small-to-medium sized, multi-level directed acyclic graph models such as those we expect in our analysis context.

Appendix A.5 provides example pseudocode of the Bayesian-based ranking algorithm.

4.4. Summary

In this chapter we have presented our fault localization method. We have described how faults propagate within the service-based systems and how our fault localization method traces the faults in order to localize the root causes. We have presented our fault localization technique, including the fault propagation model and the two ranking algorithms.

In Section 6.1.6 we describe the experimental tools we used to evaluate the method, followed by description of the implementation of the method in Java EE provided in Section 6.2.3.

The two ranking algorithms provide alternative ways in which the root cause candidates might be ordered. In Section 7.2, we investigate the impact of these, as well as impact of the accuracy of the dependence graph produced by the dependence discovery method and other factors on the accuracy of the fault localization.

5. Distributed Data Harvesting

In this chapter we present our distributed data harvesting method. We begin by describing the type of data the method transfers over the network. We then describe the architecture of the method, the synchronization algorithm and how it transfers the data over the network.

5.1. Introduction

The dependence discovery and the fault localization methods presented in Sections 4 and 3 respectively, require capability to harvest the monitoring data from nodes hosting services involved in the analyzed conversations. In these methods, we have taken the assumption of full availability of the monitoring data on demand over the network. However, in real world MANET networks, the methods cannot rely on availability of the data on demand due to the low connectivity (i.e. lack of end-to-end paths) between the network nodes. The monitoring data are usually hosted on nodes several hops away from nodes collecting the monitoring data. Thus, by attempting to harvest the data directly on demand, the methods would receive only incomplete subset of the data needed to build the dependence graphs and to perform the fault localization. In Section 7.2 we examine the availability of the monitoring data on demand in MANETs.

In what follows, we present our distributed data harvesting method. The goal of the method is to transfer monitoring data between nodes which cannot communicate directly on end-to-end paths in networks with dynamic topology. To transfer the data, we use an *epidemic protocol* to create network wide data synchronization overlay.

An epidemic protocol is a type of algorithms inspired by spread of virus in a biological community. The protocol entities periodically interact with one another and by doing so they exchange piece of bounded size information. In general, each entity periodically chooses randomly one *peer* with whom it will initiate interaction. During the interaction, either one or both of the entities will receive some new piece of information.

The epidemic protocol possesses qualities which make it good candidate for transfer of data in networks with dynamic topology such as MANETs. Most notably, the protocol does not assume reliable communication links. As we have presented in Section 2, the existing epidemic protocol techniques do not provide the functionality required for transfer of the dependence and symptom data of our dependence discovery and fault localization methods. Hence, in this chapter we describe new type of epidemic protocol capable to transfer such a type of data.

5.2. Time Series Data Structures

In the dependence discovery method, the occurrence of dependencies is captured by the monitor. The monitor intercepts messages and maintains dependence data representing occurrences of the dependencies. The monitor maintains the history of the dependencies divided into time slots. The dependence data are stored with a sliding expiration window such that only a limited history is maintained, using a data structure representing sliding time slots. Entries for each time slot maintain Boolean data about whether or not a given dependence occurred within that time slot. Each dependence is associated with a set of those time slots, such that when the monitor detects the occurrence of a dependence, it signifies this by setting a 1-bit flag in the corresponding time slot. It also records identifying information about the source and target of the dependence. The set of the time slots is thus representing aggregated *time series* of dependence occurrence events. As shown in Figure 3.3, the set of time slots is shifting with system run-time as new time slots are added and obsolete removed.

Similarly to the dependence discovery method, the fault localization method uses a set of distributed monitors. The monitors are deployed in service components (e.g., in a service container) and are responsible for extracting the symptoms of service failures from system logs. In the fault localization method, the symptoms of faults extracted from logs are classified into several failure modes (such as send failure or software failure). The set of various symptoms of one failure mode can be represented as a single time series of events same as the time series of dependence occurrences. The failure mode's time series can be thus aggregated into single set of time slots same as the dependence occurrence data.

The distributed data harvesting method uses the time slot data structure to transfer the aggregated time series data between nodes. The advantage of using the time slots is the efficiency of the data storage and transfer. A time slot requires only one bit to represent any number of events occurring within the time slot period. However, the negative side effect is certain loss of precision due to the aggregation of the data leading to a higher rate of false positives. The size of the time slot is thus controlling the rate of false positives as well as it affects the network overhead caused by transferring the data.

5.3. System Architecture

The goal of our distributed data harvesting method is to transfer data between nodes which cannot communicate directly on end-to-end paths. The method uses set of distributed *synchronization agents*, as illustrated in Figure 5.1. The synchronization agents are deployed on network nodes and are responsible for synchronization of data in epidemic fashion (i.e. by gossiping) with neighbouring nodes and for maintaining backups of received data. The agents pass on local dependence and symptom data to its neighbours, maintain backups of data received from other agents and pass on data received from other agents. Thus creating network wide data synchronization overlay. The dependence discovery and the fault localization methods use the local backup stores instead of to request dependence or symptom data on demand.

5.4. Data Synchronization Algorithm

The distributed data harvesting method employs a synchronization algorithm to propagate the data throughout the network. The synchronization algorithm is shown in Figure 5.2 and consists of the following high-level steps:

1. *Synchronization cycle.* On each node, the data synchronization is executed repeatedly in cycles. In each cycle, all data synchronization steps are executed. In the the Figure 5.2 the cycle starts with the “Start synchronization cycle” and ends with the “Wait till next cycle”.

2. *Peer selection.* First step of the synchronization cycle is to select *peer* to which data will be sent. The algorithm uses certain criteria in selection of the peers, in order to maintain high success rate of sending the data over unreliable links as well as to maintain randomness in the selection process which is an important aspect of efficient data dispersion.

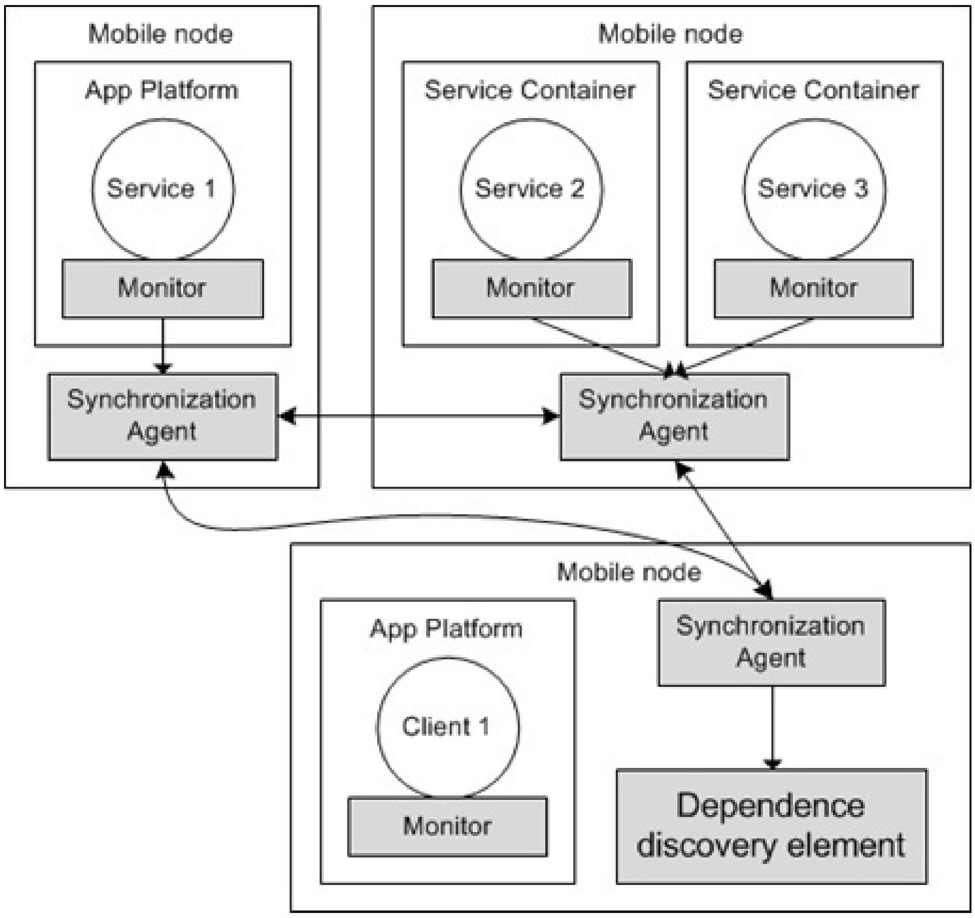


Figure 5.1.: Architecture of distributed data harvesting

3. *Calculation of dataset to transfer.* For each peer, dataset to be sent is calculated based on changes in the data since the last successful synchronization with the peer.

4. *Send dataset.* The dataset is stored into a space efficient data structure and sent over the network to the peer.

5. *Update last synchronization timestamp.* The network links are unreliable and reception of the dataset by the peer is not guaranteed. Therefore, for each peer the algorithm maintains information about which data were already successfully sent. Hence, in next round only new changes (since the last successful synchronization) are send to the peer.

In the following sections, we describe the steps of the synchronizaiton cycle in detail.

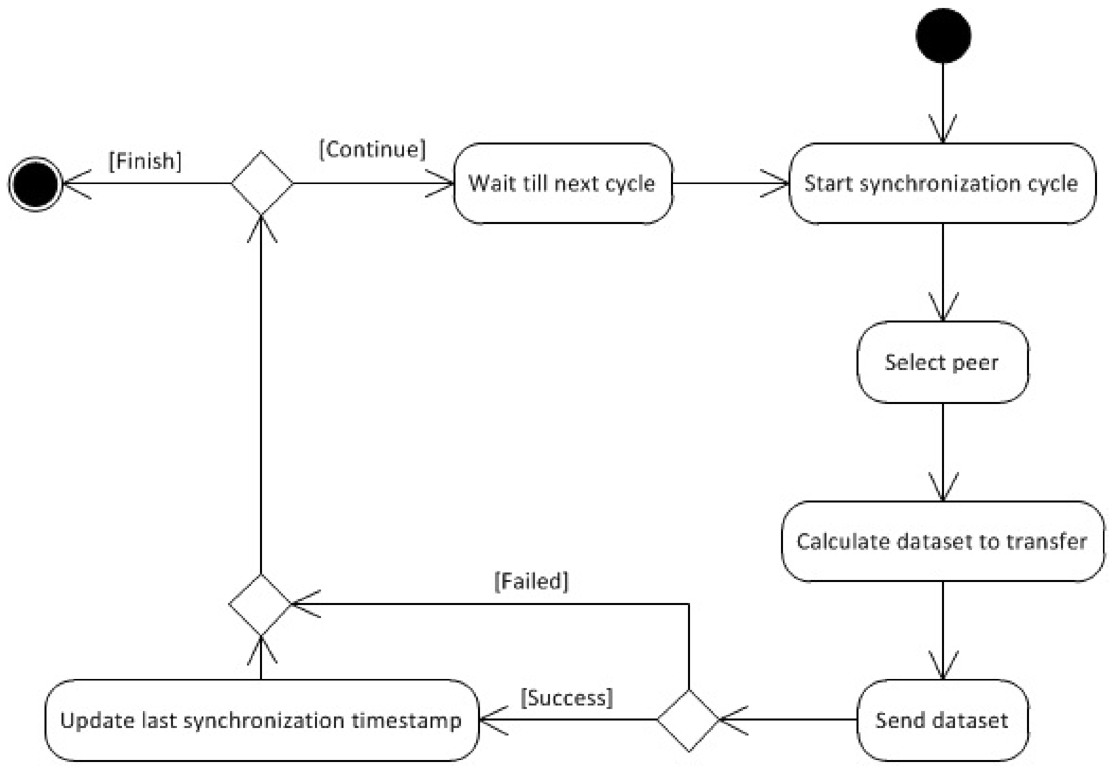

[Finish]
[Continue]
Wait till next cycle
Start synchronization cycle
Select peer
Calculate dataset to transfer
[Failed]
[Success]
Update last synchronization timestamp
Send dataset

Figure 5.2.: The data synchronization algorithm

5.4.1. Peer selection

The goal of the peer selection is to ensure the even dispersion of the data across the network in a simple yet effective manner. To this goal, we use a *random*, *memory-less* peer selection process, i.e., the peers are selected randomly by each agent regardless of to whom and when the data were sent before. The random selection is employed because, in mobile networks, it is impractical to maintain a structured overlay topology to disseminate information across the network; such a random strategy has been proven effective for information dissemination in dynamic networks [44]. Furthermore, it is a memory-less process since, when data are disseminated in a fully distributed manner, keeping track of who has received which piece of data in the network is very costly (if not impossible), particularly in dynamic networks like MANETs.

Specifically, at the beginning of each synchronization cycle, each agent first determines a candidate set of agents for the targets of the data synchronization in that cycle. The candidate set is determined such that it contains those with high probability of successful data transfer. In MANETs, as the hop-distance between two nodes increases in the network, the packet delivery ratio between those nodes decreases dramatically. Hence, we use the hop distance, which can be conveniently obtained from the local routing table, as the criteria for selecting the candidate set, i.e., a peer agent is put in the candidate set if the hop distance from the source is within a certain threshold (we use 1-hop threshold in our experimental results). Once the candidate set is determined, then a random subset of nodes from the candidates is selected as the actual peers to be sent the data by the source agent. The number of the peers selected is bounded by a configuraton parameter.

The selection mechanism is illustrated in Figures 5.3 a and b. In this example, the distance threshold for nodes to be included in the candidate set is set to 1 hop. The upper bound of the number of peers is set to 2.

In the first cycle shown in Figure 5.3a, the candidate set will contain all nodes directly reachable from the center node. The direct reachability is shown with the dashed circle. The candidate set will thus contain nodes 1,2,3,4. From this candidate set 2 peers 1,2 are randomly selected. In the next synchronization cycle shown in Figure 5.3b, the positions of the nodes changed and now the candidate set contains nodes 2,3,7,5,8 and the randomly selected 2 peers are 3,8. The changing topology and the random selection process ensure that the peers are selected randomly.

Appendix A.6 provides example pseudocode of the peer selection algorithm.

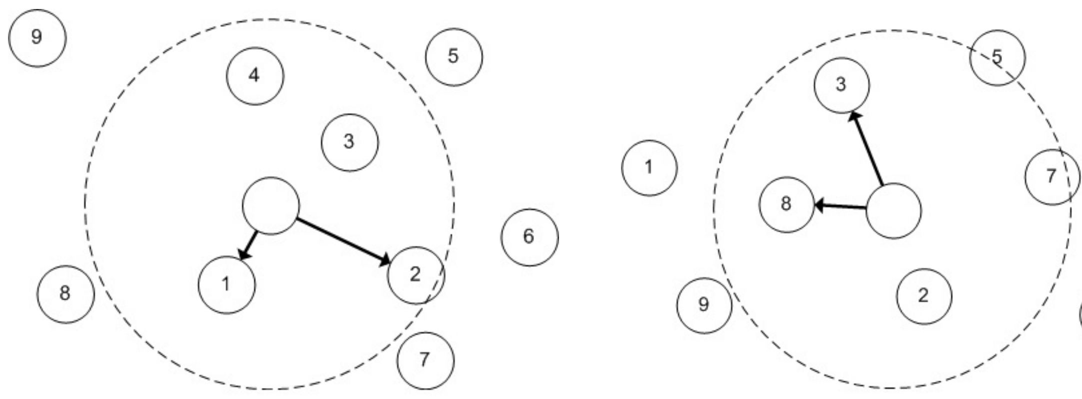


Figure 5.3.: Selection of peer in two subsequent synchronization cycles. First cycle (a) and subsequent second cycle (b)

5.4.2. Calculation of dataset

For each peer in synchronization cycle, a dataset containing all data to be sent to the peer is calculated individually. The dataset contains only incremental data changes since the last successful synchronization with the peer. The dataset includes both the data from the local monitors as well as the data from the locally maintained backup store. Unlike data from the local monitors, which are available up to date, the backup data originating at the other nodes are received with some delay. The agent therefore maintains information about the latest successfully synchronized time slot of each time series with the peer. In this way, when the dataset is calculated the agent can determine changes in the data since the last successful synchronization with given peer.

The time slots to be included in the dataset are selected based on the following criteria:

1. The set of time slots of each time series (representing either single dependence or failure mode) is examined individually.

2. Included are only time slots, which were not yet successfully sent to the peer. This is ensured by recording timestamp of the latest time slot of each time series successfully sent to the peer. Thus in each synchronization cycle are included only incremental changes since latest previous successful synchronization.

3. The included time slots must be no older than certain maximum limit. The limit is an configuration parameter of the method. When first synchronization between pair of nodes occurs, this parameter limits the inclusion to only relevant recent data. Moreover, due to dynamic network topology as well as due to the randomness of the peer selection, the period of time between subsequent synchronizations of any given pair of nodes may be long enough to lead to potential inclusion of obsolete data. Hence, this parameter ensures than only recent data are propagated across the network. In this way, the parameter minimizes the network overhead by eliminating obsolete data from the synchronization process.

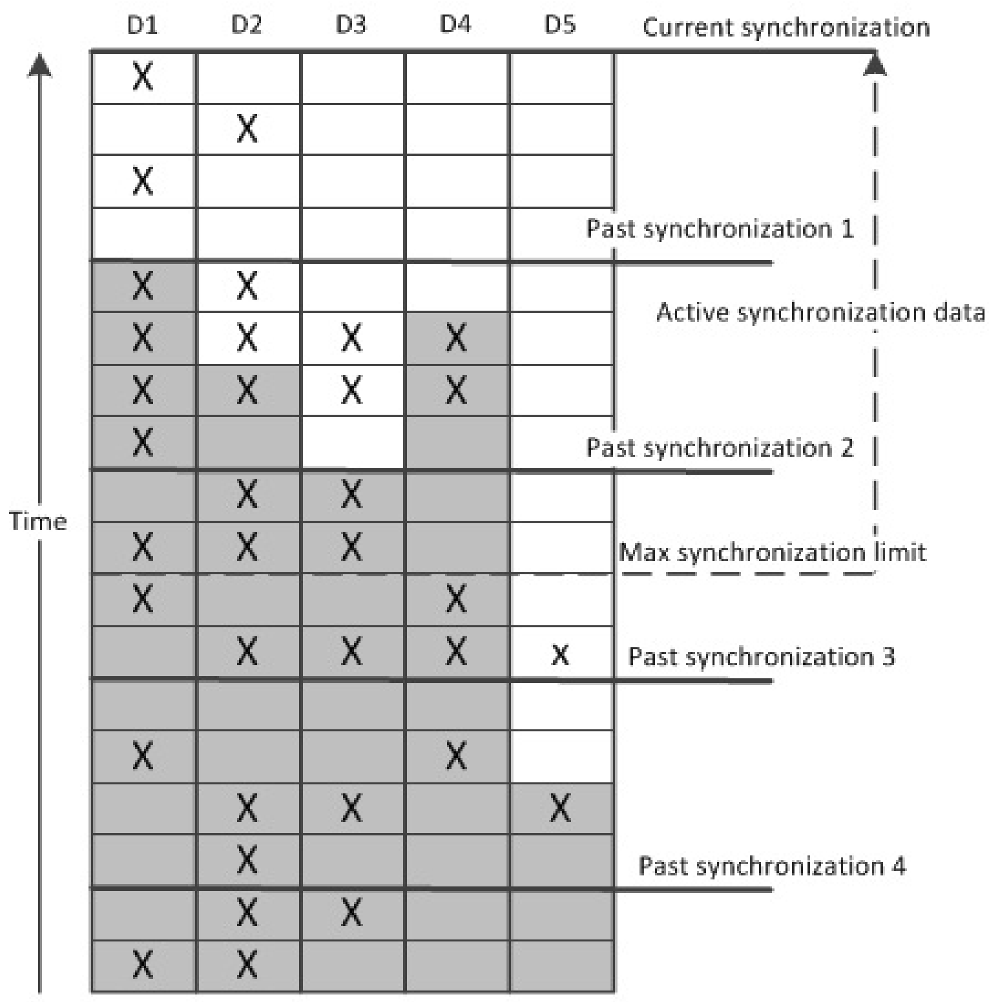
4. Only time slots with value 1 are important to be transferred. The goal of the method is to transfer evidence about occurrence of events. The evidence of lack of events is not needed to be transferred since it is implied by lack of data. Thus, only time slots with value 1 are targeted for the transfer. Nevertheless, not all time slots with value 0 can be eliminated from the transferred dataset; instead, the transferred data include all time slots with value 1 and time slots with value 0 occurring between them in order

to allow correct reconstruction of the set of time slots at the peer. This mechanism is a from of data compression and further decreases network overhead by minimizing the amount of data to transfer.

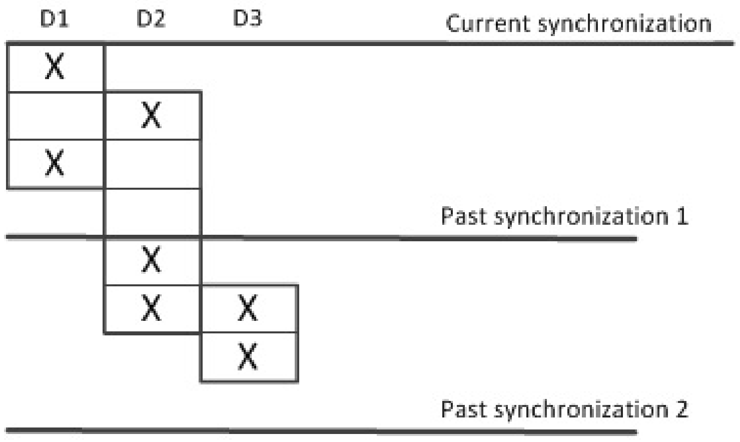
In Figures 5.4 a and b is ilustrated example of selection of time slots to include in the dataset. In the Figure 5.4a is a set of 5 dependencies, each consisting of set of time slots. The grey color marks time slots which were successfully sent to the peer in some of the previous synchronization cycles. The timing of the previous synchronization cycles is shown as a horizontal line across the set of time slots depicting the time of the synchronization relatively to the start and end of the time slots.

In the Figure 5.4b is shown the set of dependencies and time slots extracted from the Figure 5.4a based on the selection criteria. The dependence D1 has three time slots, first and last with values 1 (X) and the middle time slot with value 0 is included in order to allow the correct set reconstruction. The dependence D2 contains slots from which some are older than the last synchronization cycle. This is because the data, although representing events older then when the synchronization cycle occurred, were received from some other node later after the synchronization cycle. The dependence D3 consists only of slots older than the last synchronization cycle. The dependence D4 does not contain any new time slots with value 1 and thus no data are transferred. The dependence D5 does not contain any time slots to transfer because the data not synchronized yet are already older than the configured maximum limit for the data to transfer and thus are eliminated from the synchronization.

Appendix A.7 provides example pseudocode of the dataset calculation algorithm.



(a)



(b)

Figure 5.4.: Selection of time slots to include in the dataset to transfer. Locally stored data (a) and extracted dataset to transfer (b)

For each time series of the transferred data, the dataset contains two pieces of information. First is a timestamp of latest time slot of the transferred set of time slots. Second is a series of bits representing values of the time slots. These two pieces of information allow the peer to reconstruct the set of the time slots and to merge it with the backup store.

5.4.3. Transfer of dataset

After the dataset is calculated, it is sent to the peer. The dataset is transferred in a single message or transaction. Since the network links are unreliable, the agent waits for response message confirming successful reception of the data. Only when confirmation of the successful reception is received, the agent updates the timestamps of the latest successfully synchronized time slots with the peer.

If network failure prevents sending the dataset or if the confirmation is not received on time and the response timeout expires, the agent stops waiting for the response and does not update the timestamps of the latest successfully synchronized time slots. Consequently, in the next synchronization cycle with the peer, the data, which were not successfully sent will be included in the new dataset and sent again.

The response timeout period of the confirmation is a parameter of the method. The response timeout should not be longer than the frequency of the synchronization cycle. If so, it may lead to race condition and overwriting of the timestamps of the latest successfully synchronized time slots. The agent may then repeatedly attempt to send data which could have been already received and thus will increase the network overhead.

Appendix A.8 provides example pseudocode of the dataset transfer algorithm.

5.4.4. Reception of dataset

When the agent receives the dataset, it stores the data into its local backup store. The received data of each time series are combined with the corresponding locally stored time series backup data. The data are combined such that all received time slots with value 1 are copied into a corresponding time slot. The time slots with value 0 are disregarded as explained above in Secion 5.4.2. The backup store data structure maintains all data for limited period of time. Obsolete data are regularly pruned. In this way the amount of data stored is limited to minimum.

5.4.5. Synchronization cycle

The synchronization cycle is repeatedly started by the agent. The start of one cycle does not depend on the time of end of the previous cycle. Thus even though each cycle has certain duration of time the agent will start all cycles in precise time intervals. In this way the agent ensures that the data are dispersed at regular time intervals regardless of the amount of data passed on, quality of network links and other variables which may cause delays.

In this chapter we present our distributed data harvesting method. We begin by describing the type of data the method transfers over the network. We then describe the architecture of the method, the synchronization algorithm and how it transfers the data over the network.

5.5. Summary

In this chapter we have presented our distributed data harvesting method. We have described the type of data the method transfers over the network, the architecture of the method and the epidemic protocol which transfers the data over the network.

In Section 6.2.4 we describe the implementation of the method in Java EE.

The distributed data harvesting method is not useful in and of itself, but rather as a mechanism of data transfer used by other methods. Thus, in Section 7.3 we evaluate the method in conjunction with the dependence discovery method, and in Section 7.4 we evaluate the method in conjunction with the fault localization method. In particular, we will examine the capacity of the method to transfer data from nodes hosting monitors to nodes performing the dependence discovery and fault localization, and we investigate how the aggregation of the dependence and symptom data impacts accuracy of the two methods.

6. Experimental Toolset

In this chapter we present the experimental tools we used in the evaluation of our methods. For an initial evaluation of the dependence discovery and fault localization methods we have developed extension of a standard network simulator and analyzed the output in a database engine. In later and more advanced series of experiments, we have implemented all of our methods in Java language and Java EE platform, and used accurate emulation toolset to perform the experiments. In what follows we describe the design of the service-based system simulator used in the early evaluation, followed by description of the emulation tools and architecture of the implementation of our methods used in the later evaluation.

6.1. Service-based Systems Simulator

For an evaluation of our dependence discovery and fault localization methods, simulator which can closely replicate the behaviors of service-based systems running on MANETs is required. However, simulation of service-based system is a difficult task because of the inherent complexity of the service-based systems as well as of the mobile networks. Service-based systems consist of some number of services that interact with one another in order to complete client requests. Each service provides a set of methods for use by other services or clients. A method may use any number of other methods provided by other services to carry out its functionality. Thus, services are interconnected with each other. Furthermore, clients and services discover other services to invoke dynamically at runtime. Thus, service interconnections are unstable, particularly in mobile networks. Clients initiate the flow of service requests by sending messages that request method executions, and then wait for some response.

Existing simulation tools fall short of providing such capabilities. Packet-level network simulators, such as NS-3[1] and QualNet[2] provide detailed implementations of mobile, wire-

[1]http://www.nsnam.org/
[2]http://www.scalable-networks.com/products/qualnet/

less networks, but lack the ability to replicate complex behavioral aspects of service-based systems. These aspects are addressed in high-level service simulators [57], which unfortunately do not provide a means to simulate a complex network layer.

In this section we introduce our new tool for simulation of service-based systems hosted on MANETs. With the system and behavioral models of services built on top of a packet-based simulator, our approach allows the replication of various critical aspects, such as the cascading flows of messages in complex conversations, comprehensive client-driven workload profiles, and the propagation of faults through services. Furthermore, the simulator provides generic and easily extended models that can be used to capture modern service-based platforms, such as SOA, operating in MANET or hybrid networks.

6.1.1. Architecture

The simulator engine is built on top of the discrete event network simulator NS-3, extended with additional higher-level abstraction layers and components for simulating service entities and their interactions. NS-3 provides a comprehensive network simulation with detailed implementation of low-level network protocols. However, NS-3 provides only a simple mechanism for simulating the flow of packets from point to point. At the highest abstraction level, NS-3 provides sockets and packets as a basic network data transfer mechanism.

Figure 6.1 illustrates the architecture of our simulator engine. The simulation engine encapsulates the socket layer into a *messaging layer* that provides the abstraction of messages exchanged between end-points. The messaging layer is then encapsulated into a *service layer* that provides abstractions for entities (services and clients) and their interconnection models. Finally, the simulator provides methods for engineers to configure the simulation scenarios and their parameters, to run the simulation, and to generate output traces. In what follows, we describe these layers, models and the current implementation.

6.1.2. Messaging layer

The messaging layer provides abstractions for exchange of messages between end-points used by service-based system entities. The messaging layer is built directly on top of the socket layer of NS-3.

The abstractions provided by NS-3 for transferring data over the network are that of sockets and packets. The NS-3 socket is an asynchronous implementation of BSD socket

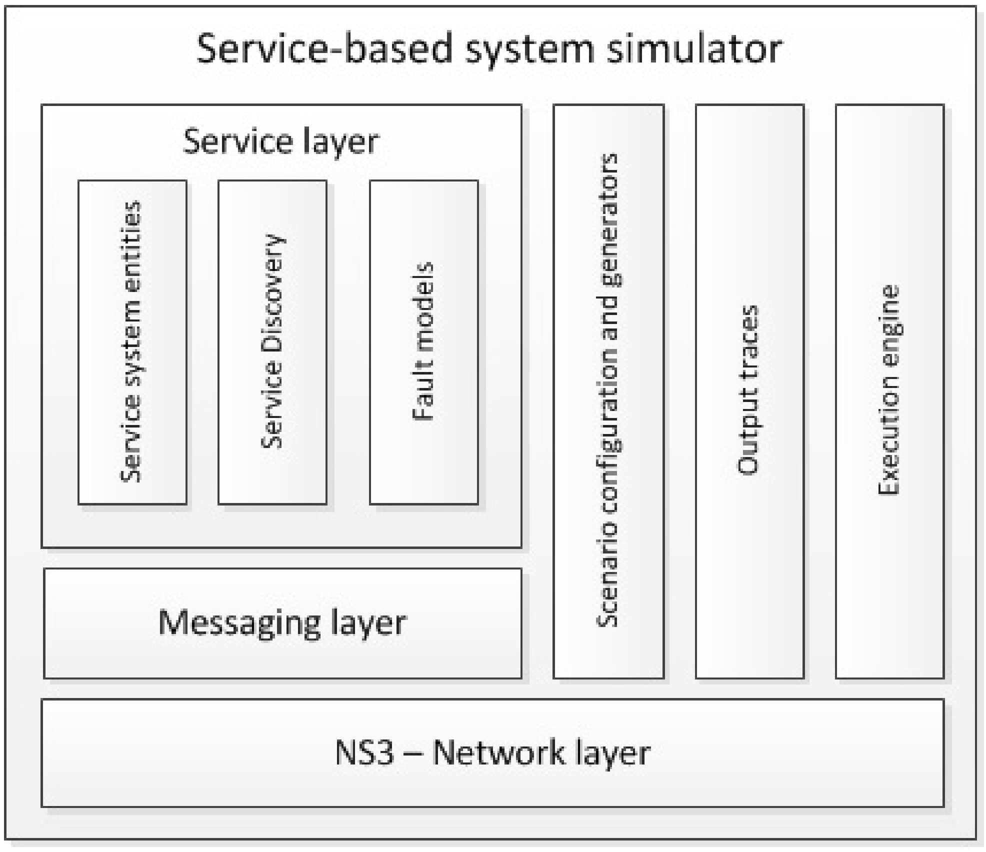


Figure 6.1.: Architecture of the NS-3 based service-based system simulator

API. The sockets are used to send and receive data in NS-3 packets. The NS-3 packet is transferred over the network in series of network packets. The NS-3 provides suit of socket types including one for each transport layer protocol implemented in NS-3 (i.e. TCP and UDP). The messaging layer encapsulates the sockets of the specific transport protocol and provides abstractions for exchange of messages.

Figure 6.2 illustrates the high level structure of the messaging layer. The messaging layer has two main objectives.

First, the messaging layer defines the abstractions used for exchange of messages over the network in unicast mode. A message represents remote method invocation mechanism such as a message in SOAP protocol. The message caries reference to the service and method to invoke and additional identification data. Additionally, the message caries certain amount of dummy data to represent the size the message would have in a real system. The messages are exchanged over the network between two types of end-points. A client end-point is used to send messages and a server end-point is used to receive messages. The client-end point is thus used by the client applications and services to send request messages to other services and to receive response messages. Furthermore, the client-end point is also used to send response messages from services back to clients and services. The server end-point is used by services to listen for and to receive invocation request messages.

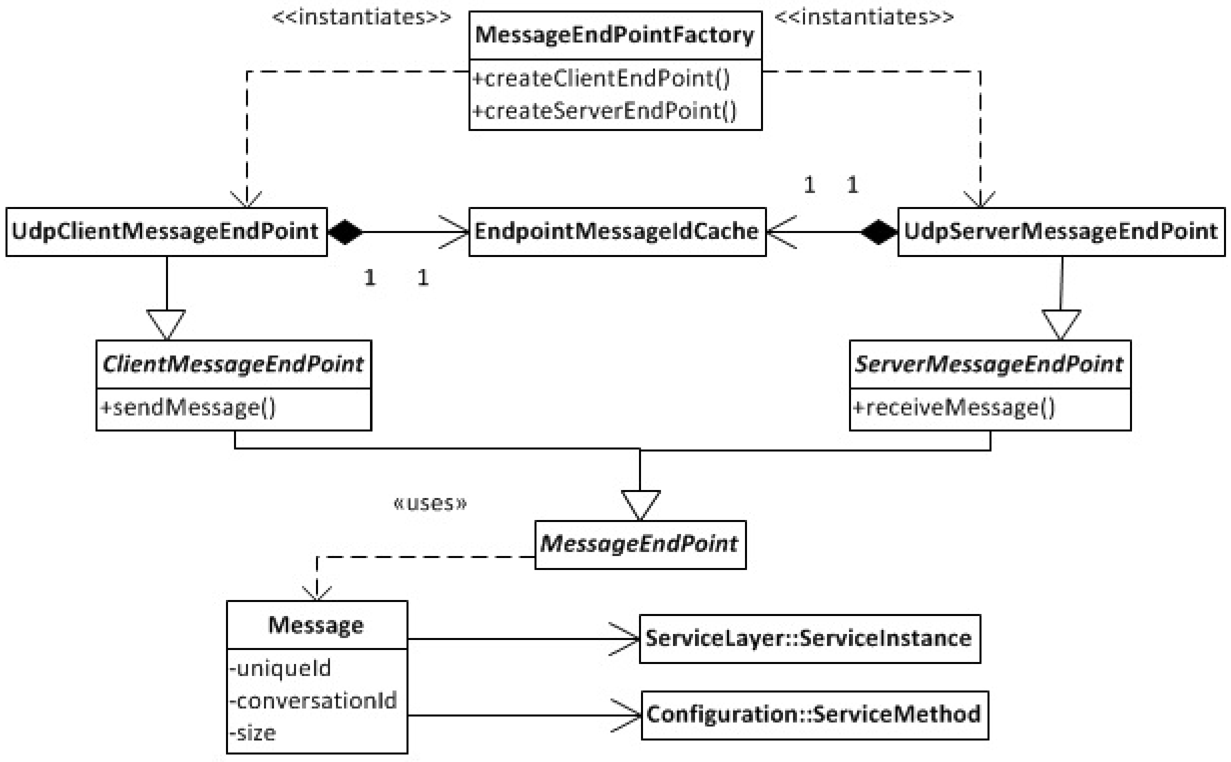


Figure 6.2.: Class diagram of the messaging layer

The messaging layer further introduces semantics into the message exchange. The end-points thus support two message exchange patterns; namely the request-response pattern and the send-only pattern. In order to support these exchange patterns the end-points report on whether a message was successfully sent between end-points or failed (i.e. send failure) and whether a response message was received on time or not (i.e. response timeout).

The second objective of the messaging layer is to provide implementation of the specific transport protocols for exchanging messages over the dynamic networks. In our current implementation, the messaging layer provides implementation of the UDP transport layer protocol. The UDP is commonly used in MANET environments because it is a lightweight stateless protocol. Thus, it allows sending data between nodes without maintaining connection or other mechanisms which are problematic or ineffective in networks with dynamic topology. However, the UDP protocol is unreliable because it does not guarantee delivery of packets or notification of failure. Thus, the UDP based end-point implementation provides additional lightweight mechanism for semi-reliable message delivery. This mechanism also provides the necessary functionality for the end-point reporting of status of message exchange. The protocol uses acknowledgement messages (ACK) to verify if payload message was successfully received.

The acknowledgement protocol works as following: an end-point will wait for certain period of time after it sent payload message to receive ACK message. If the ACK message is received in predefined period of time, the end-point will confirm successful sending of the payload message. However, if the ACK message was not received, the end-point will re-send the payload message again and again wait of the ACK message. The re-send cycle will be repeated for certain number of times. On the receiving side, as a response to any payload message received, the end-point will automatically send ACK message. The acknowledgement protocol is illustrated in Figure 6.3 for client end-point and Figure 6.4 for server end-point. The client end-point is used for either to send request message and wait for response or to send single message without response.

Due to peculiarities of the dynamic networks, the messages get frequently lost between nodes. Hence, using the acknowledgement and re-send mechanism may lead to repeated sending and receiving of the same messages. To eliminate consequences of such a condition, the UDP end-points use two strategies. First, whenever end-point receives payload message, it always sends back ACK message to confirm reception of the payload message (even if the message was already received before). Second, the end-point uses mechanism of dropping redundant payload messages already received i.e. preventing redundant invocation of services. This is an important feature ensuring integrity of the computations, required in environment built on top of non-reliable transport protocols.

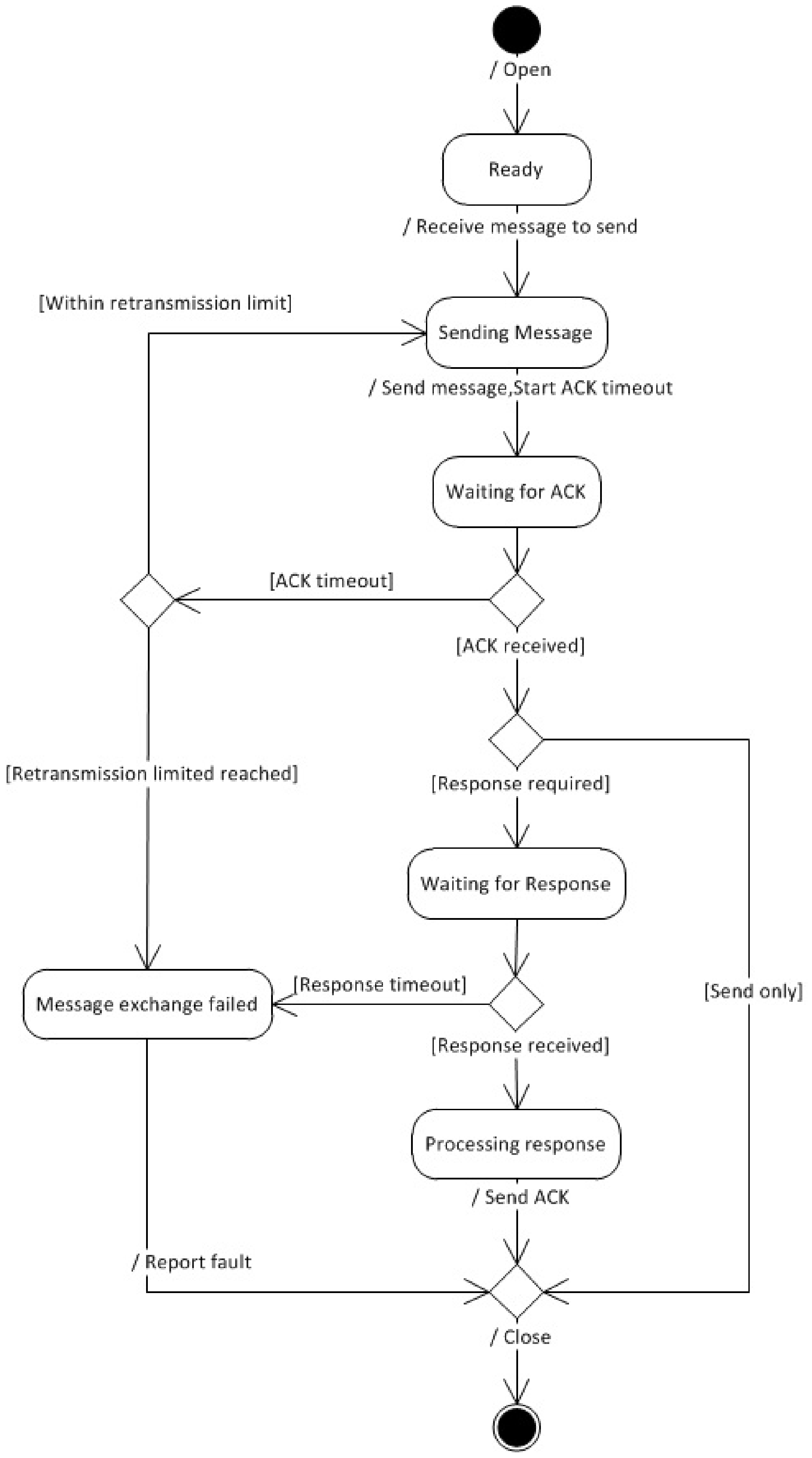


Figure 6.3.: State-machine diagram of the UDP client end-point protocol

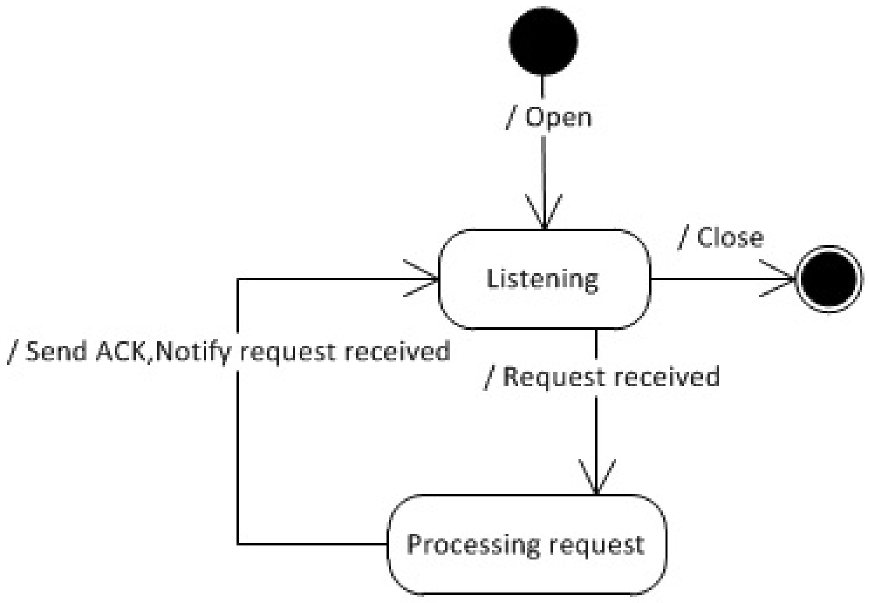


Figure 6.4.: State-machine disgram of the UDP server end-point protocol

6.1.3. Service layer

The service layer of the simulator consists of several abstraction models: entities, interconnections, workloads, faults, messages and deployment.

(1) *Entity model:* Entity models provide the building blocks of the service-based system simulation.

- *Clients* represent applications used by end users. Each client behaves as an autonomous entity that contacts a set of services at times (random or deterministic) configured by the engineer.
- *Contracts* represent definitions of interfaces of services. Each contract defines an interface of a type of service as a set of methods. A contract is provided by (i.e. implemented by) one or more services.
- *Services* represent autonomous self-contained functional units. Each service adheres to (i.e. implements) one contract and has a set of methods that are available to be used by clients and other services. Each method contains an abstract definition of its computation consisting of delays to simulate processing time, and a set of steps that send requests to other services.

In the Figure 6.5 are illustrated the configuration elements of the service-based system entities. Furthermore, in the Figure 6.6 are shown the runtime elements of the service-based system entities. In the Table 6.1 are provided typical values used in configuration of the Entity model in our experiments.

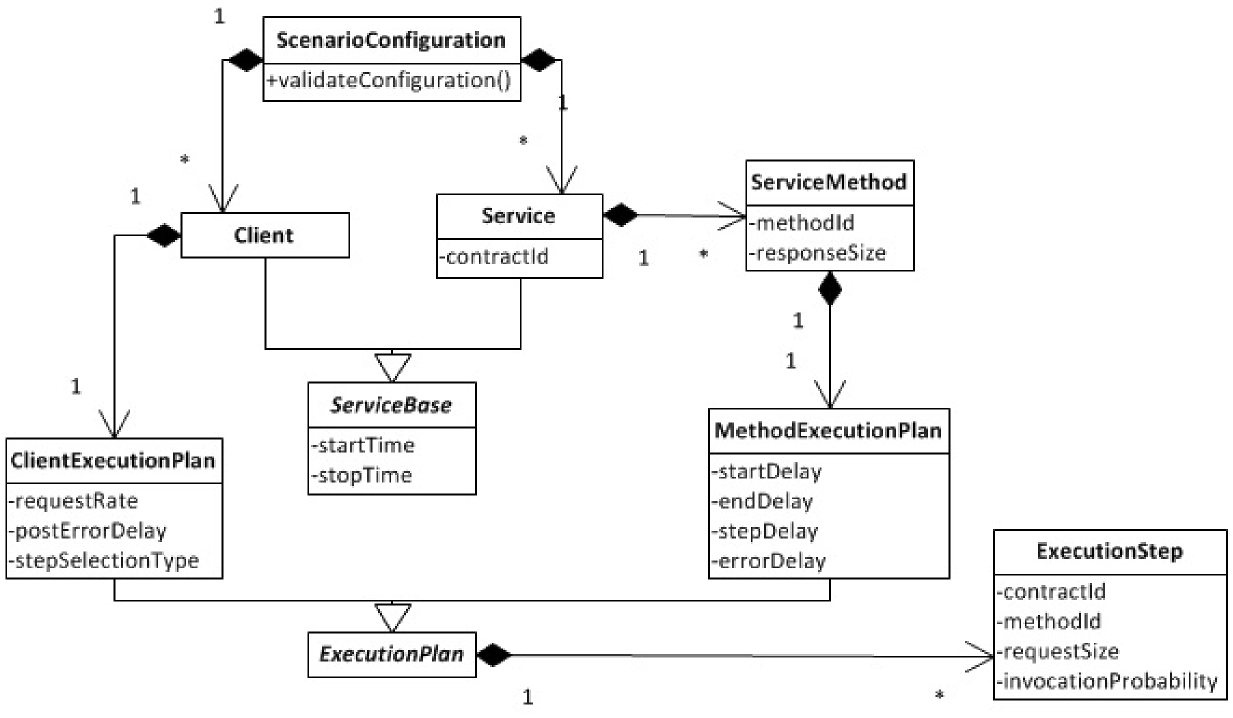


Figure 6.5.: Class diagram of the configuration elements of the service-based system entities

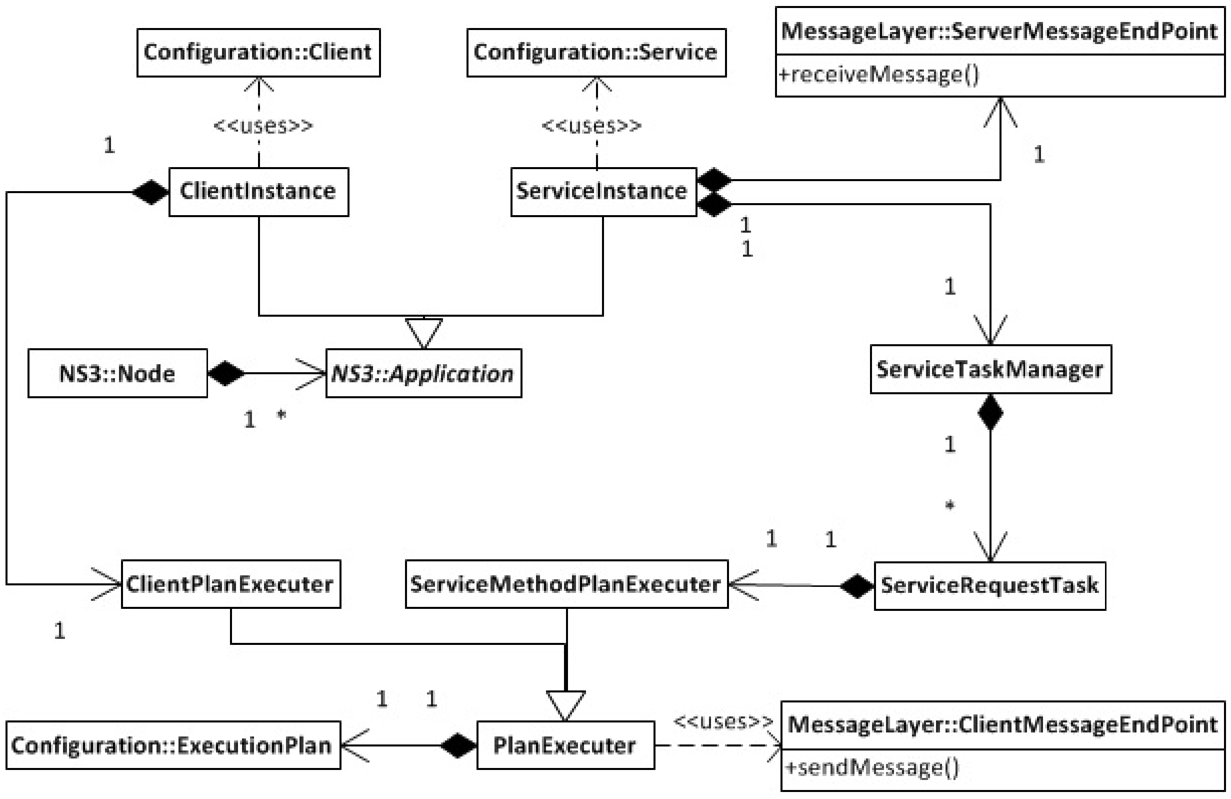


Figure 6.6.: Class diagram of the runtime elements of the service-based system entities

(2) *Interconnection model:* The interconnection model defines the methods in other service contracts with which each entity in the service-based system interacts (i.e., sends service requests and receives responses). Two types of interconnections are defined: client-to-service, and service-to-service. The simulator provides probabilistic as well as deterministic generators of the service-based system interconnections. The probabilistic generator creates a randomized configuration with predefined connection probabilities. The deterministic generator allows the engineer to have control over the specific interactions in the system.

The interconnection model also defines the *service discovery* mechanism used during the system runtime. The service discovery mechanism provides functionality to the clients and services to discover service instances to send messages to during the system runtime. It is an essential component in service-based system where interconnections are defined between entities and contracts (i.e. types of services) but not services themselves. Thus, during the system runtime, the entities have to discover which actual service instance they should interact with based on the contract the service instance provides. This task is particularly important in MANETs with the dynamic topology continuously altering connectivity between nodes and consequently the availability of services.

In general, before every request, service-based system entity will query a service registry for a service instance to send the request to. The service registry will select the most appropriate service instance based on its availability or some other metric. The simulator provides three types of service discovery mechanisms; based on physical distance, based on metric from routing tables (i.e. hop distance) and based on fixed configuration. Figure 6.7 illustrates the service discovery components.

The interconnection model allows configuring specific system topologies. For example, a frequently used topology in the service-based systems is a 2-tiered topology. In this type of system, the first tier consists of the connections between the clients and a set of "front-end" services, while the second tier consists of the connections between the services themselves. In our experiments we make frequently use of the specific service topologies.

In the Table 6.1 are provided typical values used in configuration of the Interconnection model in our experiments. In this example configuration, the interconnections are defined probabilistically.

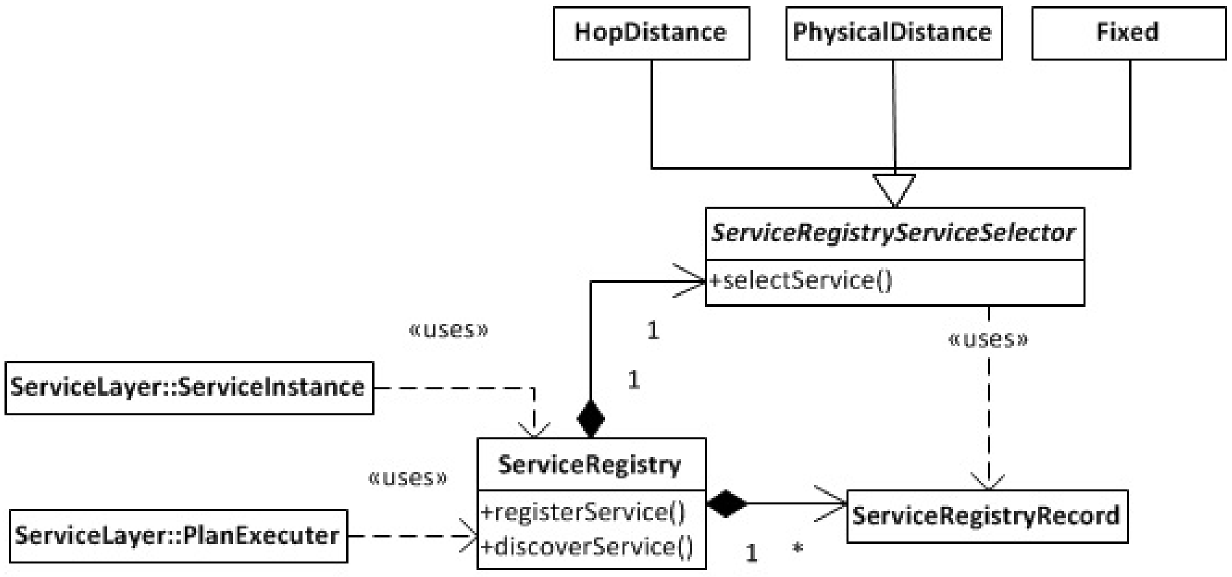


Figure 6.7.: Class diagram of the service discovery components of the service-based system

(3) *Message model*: There are three types of messages exchanged between entities: requests, responses, and exceptions. Request messages are used to invoke methods in other services, while response messages are sent by services back to the requesting entity upon

the completion of the requested method. Exception messages are used to propagate fault symptoms caused by network or service faults. The flow of messages exchanged between services during the processing of a client request is called a *conversation*. In the simulator, all messages contain information about the conversation to which they belong. The conversation information is designed to replicate the behavior of WS-* standards such as WS-Addressing.[3]

(4) *Fault model*: Services running on MANETs are exposed to potentially frequent faults in the network due to network instability and in the services themselves due to resource constraints (and bugs). While network failures are immediately captured by the network simulation layer, we must include a service fault model that defines the failure behavior of the services.

The network faults are configured directly within configuration of the physical network. In MANETs, the essential configuration of the network includes the propagation loss model which defines the quality of wireless links between nodes. The simulator provides series of deterministic and probabilistic service fault models as well as the capability to configure composite fault model behavior. The fault models are injected into the services and into the service methods to allow fine grained configuration of the fault behavior. Figure 6.8 illustrates the components of the fault model.

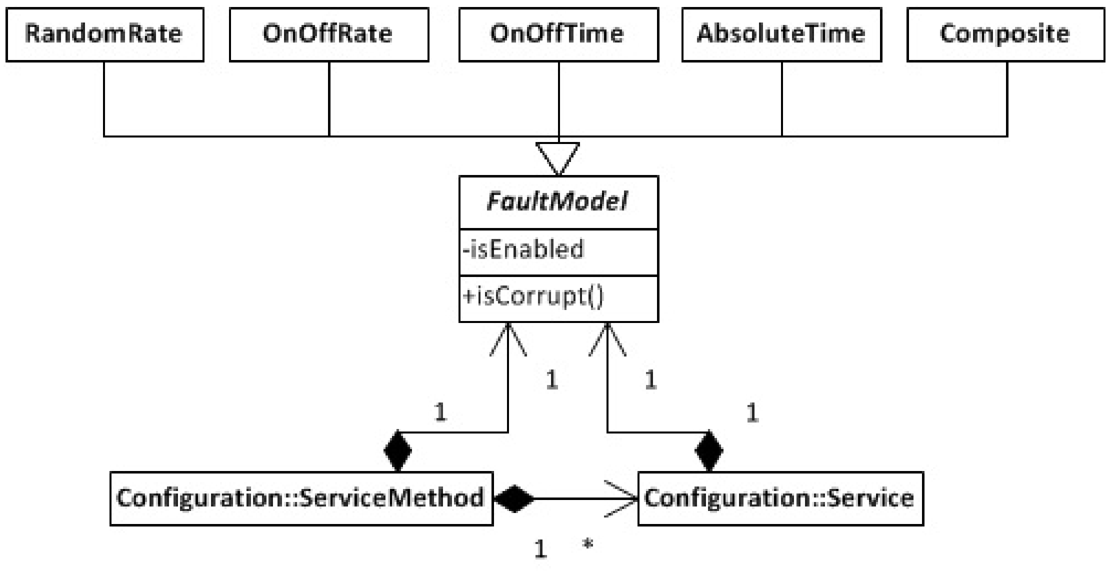


Figure 6.8.: Class diagram of the fault model components of the service-based system

[3]http://www.w3.org/Submission/ws-addressing/

In the Table 6.1 are provided typical values used in configuration of the Fault model in our experiments. In this example configuration, we use a probabilistic On/Off fault model to fail the services.

(5) *Workload model*: In service-based systems the workload is initiated by clients sending requests to services. The workload model defines the rates of such requests. In our current implementation, clients repeatedly, and at pre-configured random times select one method to request out of the set of available service methods, and then waits for a response.

In Figure 6.9 is shown client workload algorithm. The simulator provides two techniques to select a method to send the request to. The method is either selected randomly from set of available methods or the method is selected based on individual probability of each method to be invoked. For each client, the set of the available service methods is defined by the interconnection model.

Upon reception of a request, the requested service method is invoked and depending on the configuration of interconnections, further messages will be sent to other services. In Figure 6.10 is shown the processing of a request by a service.

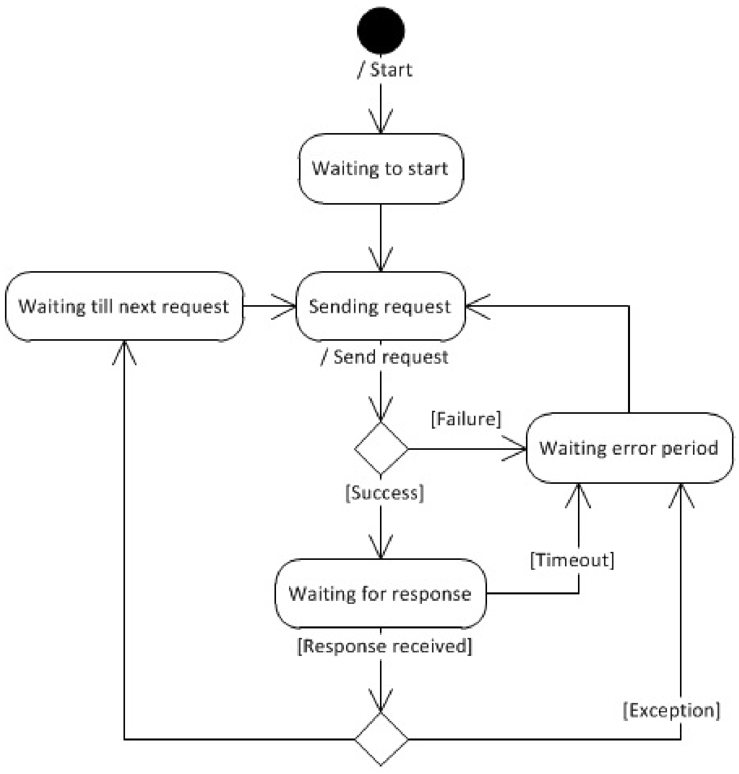


Figure 6.9.: State-machine of the client workload algorithm

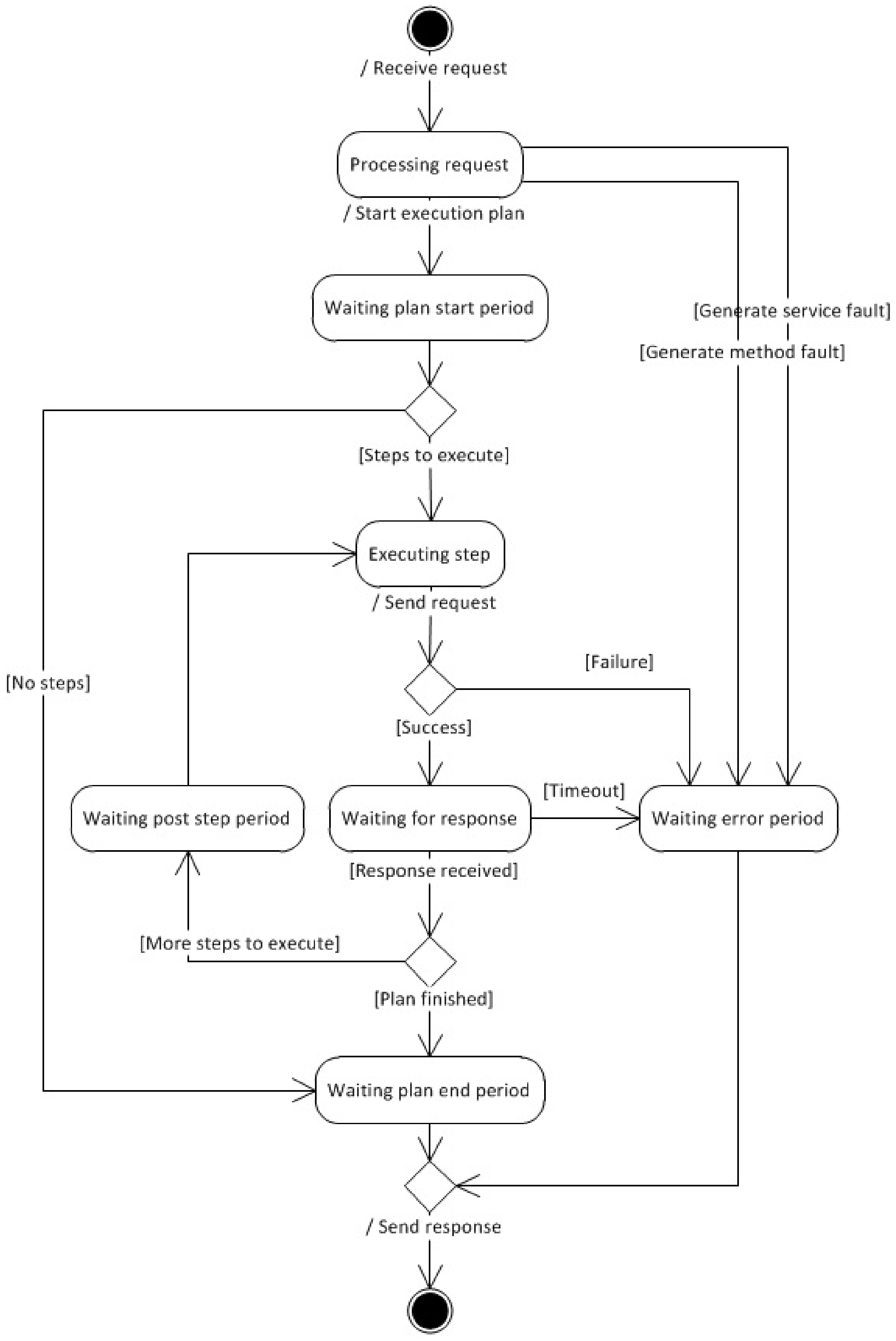


Figure 6.10.: State-machine of the service workload algorithm

In the Table 6.1 are provided typical values used in configuration of the Workload model in our experiments. In section *Workload model - client* are defined parameters of client request rate and the selection of the service method to send request to. In section *Workload model - service* are defined parameters of delays representing configuration of processing of requests by services.

(6) *Deployment model:* The deployment model specifies the mapping between physical network nodes and the instances of entities of the service-based system (i.e. clients and services). The simulator provides probabilistic as well as deterministic deployment methods.

In the Table 6.1 are provided typical values used in configuration of the Deployment model in our experiments. In this example, on each node is deployed one client and the services are distributed randomly across all of the nodes.

6.1.4. Simulation scenarios

The simulation scenarios are created by a configuration generator that creates scenario configurations based on a set of parameters of system characteristics. In addition to the network parameters configured through NS-3's configuration (e.g., number of nodes, mobility, wireless link characteristics, etc.), the services, clients, and their interactions and behaviors, including how and where the services are hosted, are configured for the above models. During the simulation run, the simulator records events, such as service message exchanges and fault symptoms, into trace files for analysis. Figure 6.11 illustrates the configuration and runtime components of the simulator.

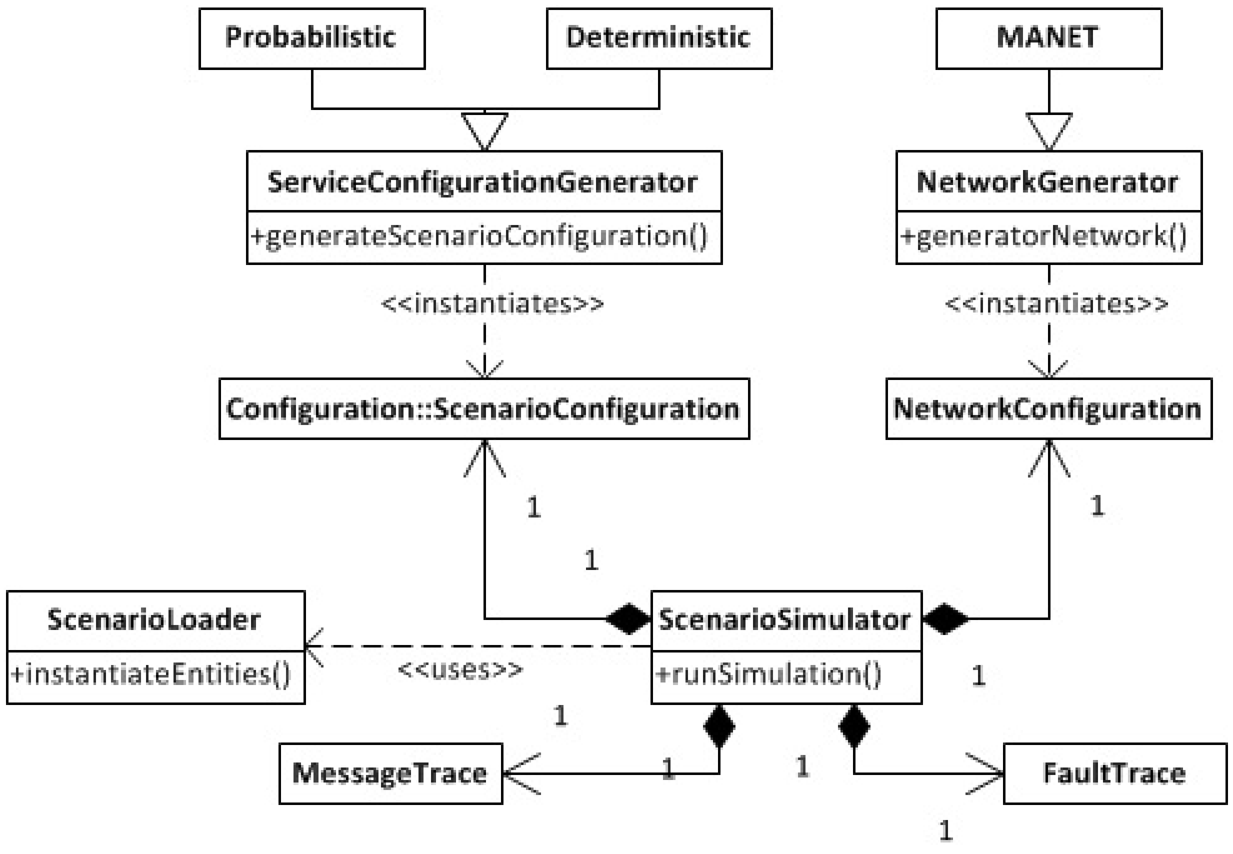


Figure 6.11.: Class diagram of the simulator configuration and runtime components

In the Table 6.1 are provided typical values used in configuration of our experiments. Aside of configuration parameters of the service layer models, the table also contains configuration parameters of NS-3 defining the underlying network layer.

Entity model	
Number of clients	50
Number of contracts	30
Number of methods in contract	2
Number of services per contract	5
Message model	
Request message size	500-1500 bytes
Response message size	500-1500 bytes
Response timeout	60 000 ms
End-point - transport protocol	UDP
ACK timeout	1000 ms
Message retransmission limit	5x
Interconnection model	
Service topology type	2-tier
Number of front-end contracts	5
Client to front-end contract method connectivity probability	0.5
Service method to contract method connectivity probability	0.01 - 0.1
Service discovery method	hop count
Workload model - client	
Request rate	5000-15000 ms
Next request after conversation fails	5000 ms
Step selection method	based on step probability
Step invocation probability	0.01 to 1
Workload model - service	
Method start delay	20
Method end delay	20
Method step delay	10
Method error delay	10
Fault model	
Service fault model	OnOffRate
Deployment model	
Client to node	1 on 1
Service to node	random
Network layer	
Number of nodes	50
Spatial bounds	75m x 75m
Mobility speed	10 m/s
Mobility model	RandomDirection2dMobility
Propagation delay model	ConstantSpeedPropagationDelay
Propagation loss model	LogDistancePropagationLoss/α3
WiFi standard	80211b
WiFi rate	11Mbps
Routing protocol	OLSR and Ipv4StaticRouting
Protocol stack	UDP/IPv4

Table 6.1.: Main simulator configuration parameters of a scenario

6.1.5. Methodology of evaluation of dependence discovery method

The evaluation of our dependence discovery method is based on the simulation framework for service-based systems operated in MANETs. As shown in Figure 6.12, the framework first generates message and fault traces according to various simulation parameters and scenario configurations. These traces are stored in a database to hold the results of the simulation runs. The dependence discovery is prototyped as a query over this database. In addition, we derive the "ground truth" used in Section 7.1 to evaluate our method.

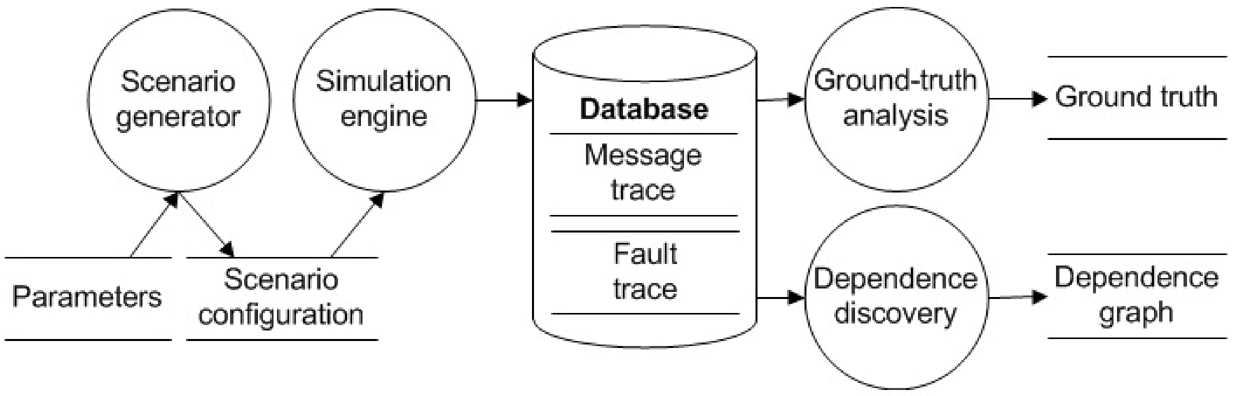


Figure 6.12.: Experimental framework used in evaluation of the dependence discovery method

The database used for prototyping of the dependence discovery is a MS SQL Server database engine. The evaluation of the dependence discovery method is implemented as a series of stored procedures which analyze records from the output trace files produced by the simulation engine. In analysis of a scenario, the output trace files are loaded into the database and ground truth of all conversations is calculated from complete record of messages exchanged between all entities within the service-based system. In next step, the prototype of the dependence discovery calculates dependence graphs for the conversations of interest. Finally, the dependence graphs are compared against the ground truth to calculate metrics used to evaluate the dependence discovery method.

Without loss of generality, in the evaluation of the dependence discovery method, we make the assumption that faults occur only at the network level, and that the services themselves are error free. From an observational point of view, any non-Byzantine service-level fault can be simulated by a corresponding network-level fault (e.g., whether the absence of a response message is due to a service fault or a network fault is indistinguishable to a service waiting for that response). Therefore, this assumption does not materially impact our results, but does allow us to concentrate on simulating the effect of the network on service interactions.

6.1.6. Methodology of evaluation of fault localization method

The evaluation of the fault localization method is built upon the evaluation of the dependence discovery method. As shown in Figure 6.13, same as in the dependence discovery evaluation, the engine first generates message and fault traces according to various simulation parameters and scenario configurations. These traces are stored in a database to hold the results of the simulation runs. Initially, dependence graphs are constructed by the dependence discovery method. Subsequently, upon the dependence graphs, the causality analysis that contributes to an FPM, and the timing-based ranking are prototyped as queries over the database. In addition, we derive the root-cause "ground truth" used in Section 7.2 to evaluate our method.

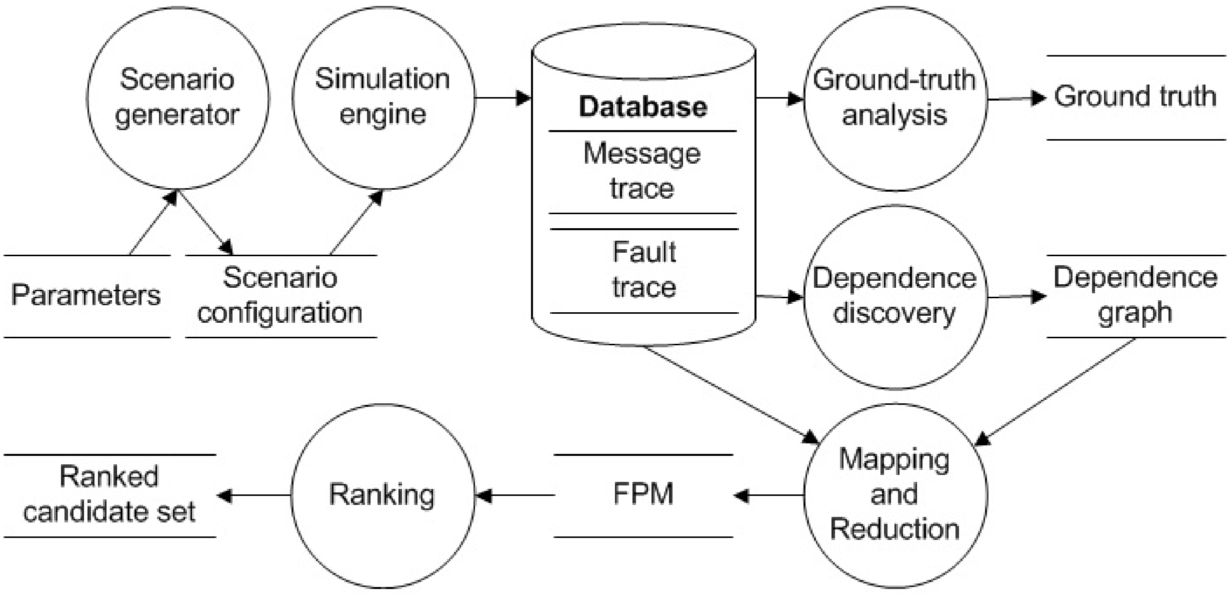


Figure 6.13.: Experimental framework used in evaluation of the fault localization method

For the Bayesian-network ranking we use MATLAB[4] with the Bayes Net Toolbox.[5] The Bayes Net Toolbox is widely used tool in experiments concerning Bayesian-networks and

[4]http://www.mathworks.com/products/matlab/
[5]http://code.google.com/p/bnt/

it provides efficient implementation of the junction-tree algorithm used in the BNet ranking algorithm described in Section 4. The failure propagation probabilities modeled in the BNet ranking algorithm are implemented as a transformation of the noisy-OR gate distribution model of the conditional probability distribution into a deterministic conditional probability tables. In this way, the inference is spatially as well as computationally efficient.

6.2. Service-based Systems Emulator

The NS-3 network simulator, underpinning the service-based system simulator described above in the Section 6.1, uses approximations of many aspects of the network behavior, components and protocols, and thus is missing some of the important network characteristics and details. The Distributed Data Harvesting method addresses problems of connectivity between nodes in MANETs and consequently its performance is sensitive to the details of the network environment. However, the connectivity between nodes in MANETs depends on combination of several network aspects, each playing important part in the overall outcome. Hence, experimental tool closely replicating the peculiarities of the MANETs is required in order to provide accurate environment for evaluation of the method.

In this section we describe the experimental tools we have used to evaluate the Distributed Data Harvesting method in conjunction with the dependence discovery and fault localization methods. The experimental tools used provide accurate and realistic environment for hosting real world service-based systems components such as the Java EE platform. In order to evaluate our methods, we have implemented the methods in Java EE and we have designed and implemented generic web services system which we use in our experiments.

6.2.1. Architecture

For evaluation of the Distributed Data Harvesting method, we have opted for combination of two tools; Common Open Research Emulator (CORE) [6] and Extendable Mobile Ad-hoc Network Emulator (EMANE)[7].

The CORE is a virtualization tool for emulation of computer networks [3, 2]. The CORE runs on Linux platform and uses native Linux mechanism of process isolation (namespace isolation) to create low overhead virtual machines. Within the CORE virtual machines is provided standard Linux operating system environment with all of the native system functionalities. The virtual machines are connected to the network provided and managed by the EMANE.

The EMANE provides real-time modeling of link and physical layer connectivity so that network protocol and application software can be experimentally subjected to the same

[6]`http://cs.itd.nrl.navy.mil/work/core/index.php`
[7]`http://cs.itd.nrl.navy.mil/work/emane/index.php`

conditions that are expected to occur in real MANETs. In the EMANE, only the two lowest levels of the OSI model stack are emulated (i.e. the physical layer and the data link layer) and on all layers above are used real network protocols provided by the Linux operating system.

The combination of these two tools provides high fidelity real-time emulation environment for evaluation of distributed systems. The setup and integration of the CORE and EMANE is a complex matter discussed by Ahrenholz et al. [4].

The emulation tools provide bare Linux operating system environment of nodes connected into the MANET network. Within the Linux environment we have built two sets of components, a generic web service system and an implementation of our methods. Figure 6.14 illustrates the architecture of our emulator toolset.

The generic web services system is built on Java platform. As an application platform we use reference implementation of Java EE, Glassfish [8]. The Glassfish is open-source software and provides full implementation of the Java application server environment. The Glassfish server is a container within which additional server components are installed. For the hosting of the web services, we use Glassfish Metro [9]. The Metro is a reference implementation of a standard Java web services stack.

The generic web services system is composed of two configurable components, a generic client application and a generic web service. The client and the web service are basic building blocks of any web services system. The functionality provided by these components is same as the functionality of the service-based system described in Section 6.1. Thus, they contain implementation of the entity, interconnection, fault, messaging, workload and deployment models. The system also stores into trace files detailed record of activities and events of the clients and web services. From this record, ground truth of dependencies and faults can be extracted in posterior system analysis.

[8] `https://glassfish.java.net/`

[9] `https://metro.java.net/`

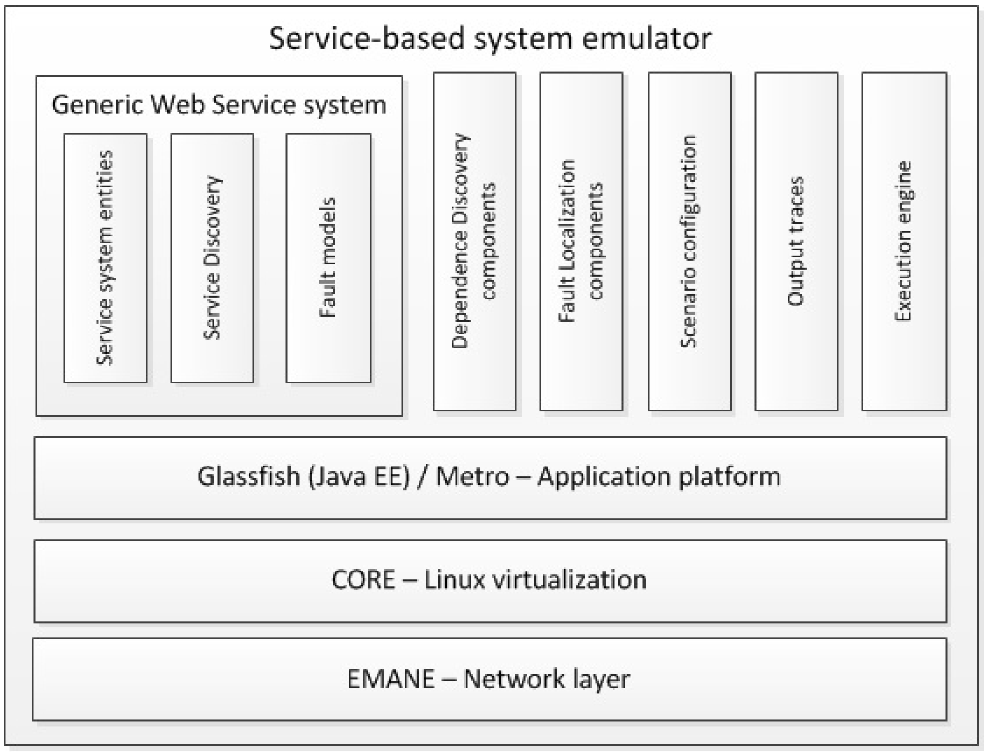

Service-based system emulator
Generic Web Service system
Service system entities
Service Discovery
Fault models
Dependence Discovery components
Fault Localization components
Scenario configuration
Output traces
Execution engine
Glassfish (Java EE) / Metro – Application platform
CORE – Linux virtualization
EMANE – Network layer

Figure 6.14.: Architecture of the CORE and EMANE based service-based system emulator

6.2.2. Implementation of the dependence discovery

The dependence discovery method is implemented in several physical components deployed on the mobile nodes. Figure 6.15 illustrates the architecture of the implementation.

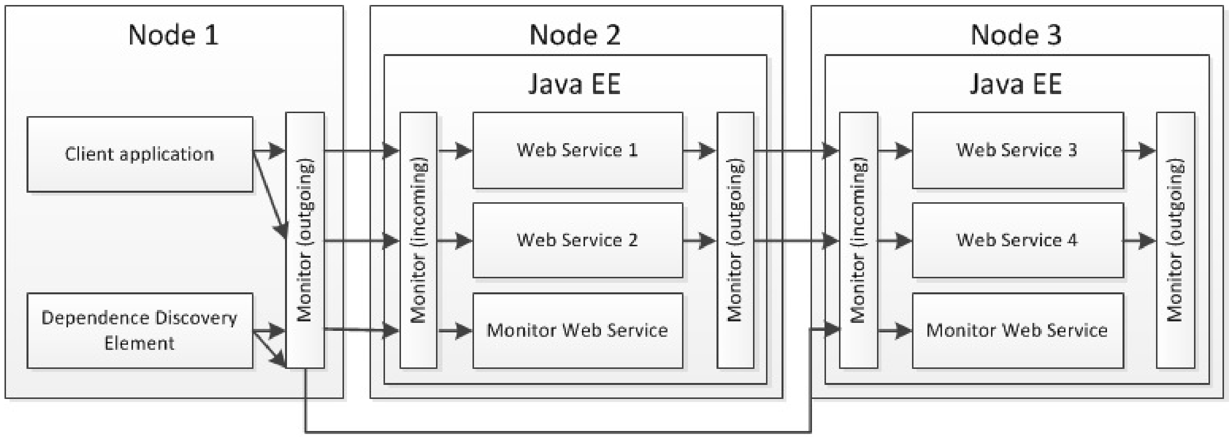


Figure 6.15.: Architecture of the dependence discovery implementation in Java EE

The monitor is implemented as a *Tube* of the Metro web services framework. The Tube is a component which is inserted into a chain of tubes which intercept and process incoming and outgoing messages of the web services and client applications. Each tube is responsible for some aspect of processing of messages between web service and network. Figure 6.16 illustrated the architecture of the monitor tube and its placement within the chain of other tubes. The monitor extracts the dependence fields from the intercepted messages and records occurrences of dependencies into a storage data structure located in a shared process memory.

The dependence data are made available for harvesting by a monitor web service. The monitor web service upon request for dependencies of a service inspects the dependence data structure located in the shared process memory and responds with list of outgoing dependencies of the service.

The dependence discovery element is implemented as a Java library. The element is used by the generic client to discover dependence graphs for conversations initiated by the client. The element uses dynamic web service end-point binding to send messages to the monitor web service hosted on the mobile nodes. The element harvests data from the monitor web services and builds dependence graphs which are stored into a trace file for further analysis.

The trace files of the generic system and of the dependence discovery produced during the scenario runtime are loaded into a database for further analysis. The analysis of the dependence discovery is prototyped as a query over this database.

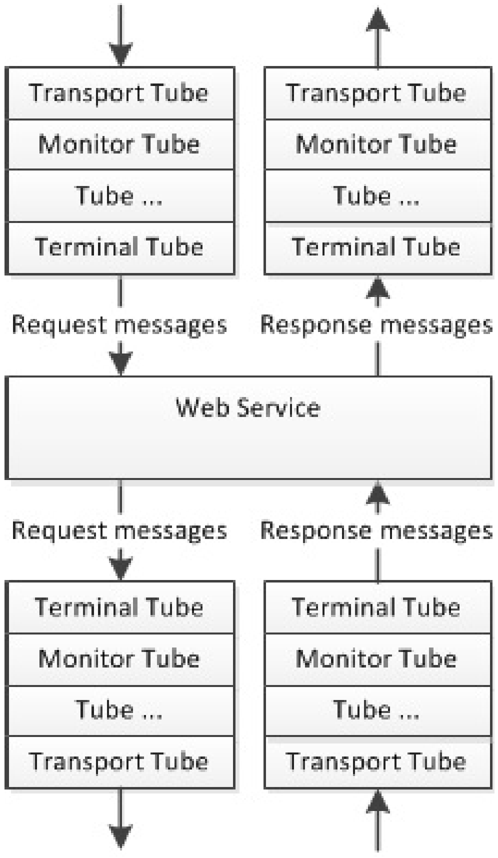


Figure 6.16.: Architecture of the dependence discovery monitor implemented as a tube in the Glassfish Metro and Java EE

6.2.3. Implementation of the fault localization

The implementation of the fault localization method is built upon the components of the dependence discovery method. Figure 6.17 illustrates the architecture of the implementation.

The generic web services report symptoms of faults into a standard Java EE server log. The symptom monitor implemented as a Java library upon request accesses the log, parses the log records and extracts the symptoms of relevant faults. The monitor web service is extended to provide the symptom data along the dependence data on demand. In this way, the network overhead is minimized.

Upon request, the dependence discovery element harvests dependencies and symptom data from the monitor web services and builds dependence graph. The dependence graph is annotated with the symptom data. The annotated dependence graphs are stored into a trace files for further fault localization analysis. After the experiment ends, the trace files are loaded into a database to hold the results of the simulation runs. Similarly as in analysis of the service-based system simulator, the fault localization is implemented as a query over the database. For the Bayesian-network ranking we use the MATLAB with the Bayes Net Toolbox.

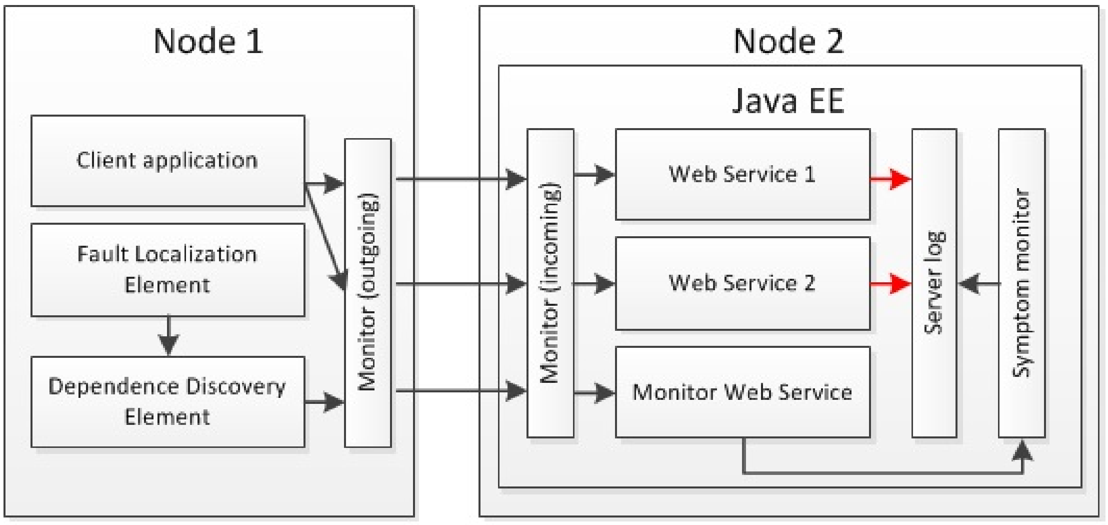


Figure 6.17.: Architecture of the fault localization implementation in the Java EE

6.2.4. Implementation of the distributed data harvesting

The distributed data harvesting method replaces the direct data harvesting mechanism and introduces new components which deliver the data synchronization functionality. The data synchronization functionality is divided between *synchronization agent* (SA) and *synchronization web service* (sws). Figure 6.18 illustrates the architecture of the implementation.

The SWS is a passive component implemented as a Java web service which resides within the Java EE server and exposes its functionality to the SA. The SWS receives data from SAs hosted on other nodes, maintains backup store and provides data to the locally hosted SA. The data received from other nodes are merged and stored into the backup store. Upon request for data from the local SA, the SWS provides combined data from the backup store (data received from other nodes) as well as data from the local monitors.

The SA is an active component implemented as a standard Java application running as an independent process on the mobile nodes. The SA is executing repeatedly the synchronization of data with other nodes. For each target node, the SA maintains information about the last successful synchronization and with this information queries the local SWS for data to be sent to neighbor nodes. The SA uses dynamic web service end-point binding to send messages to the SWS hosted on the mobile nodes.

Both of the components also record information about their activities into a trace files for analysis of their behavior and performance (such as analysis of the network overhead, success rate of synchronization attempts and others).

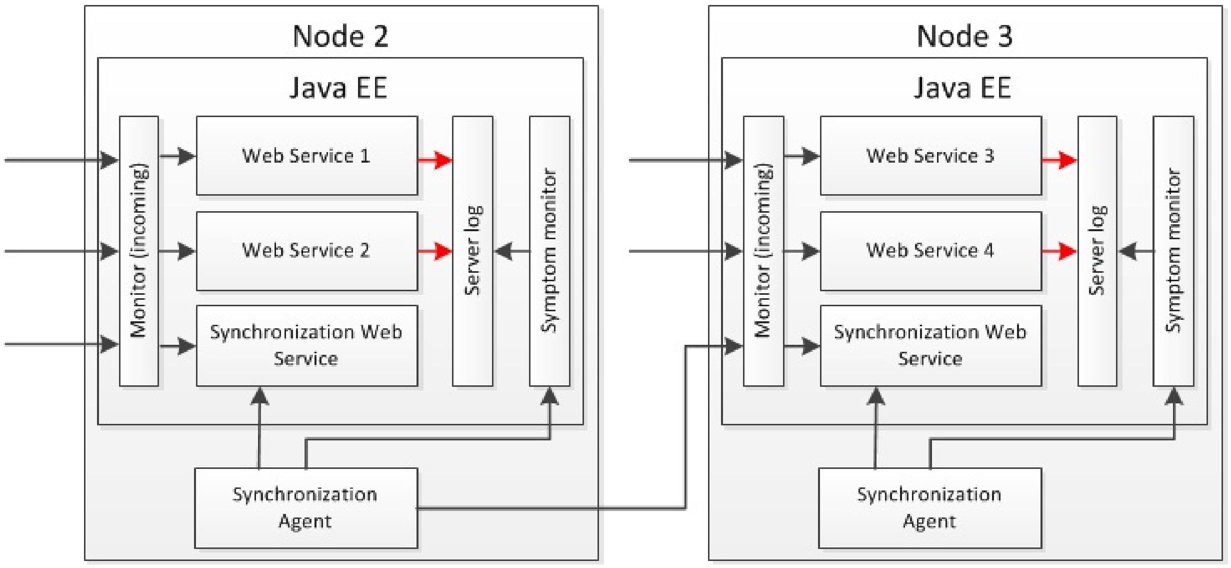


Figure 6.18.: Architecture of the Distributed Data Harvesting implementation in the Java EE

6.3. Summary

In Section 6.1 we have introduced our simulator of service-based systems hosted in MANETs. The simulator is built as an extension of a standard packet based network simulator NS-3. The simulator thus closely replicates the complex network behavior as well as the service-based system entities and models. We have used the simulator for evaluation of our dependence discovery and fault localization methods. The results of these experiments are presented in Sections 7.1 and 7.2 in the Evaluation chapter.

To our knowledge, there is currently no other comparable tool providing functionality of network and service layers simulation. Therefore, we believe that the simulator can be valuable tool for many other researchers as well. Aside of the work presented in this thesis, we have used the simulator in analysis of behavior of hybrid wireless networks presented in [62].

In Section 6.2 we have presented the experimental toolset we have used for evaluation of the distributed data harvesting method used in conjunction with the dependence discovery and fault localization methods. The toolset provides realistic emulation environment for hosting real world system components. In order to evaluate the methods, we have designed and implemented generic web service system, and implemented all of our methods in Java EE. The results of experiments based on these tools and components are presented in Sections 7.3 and 7.4 in the Evaluation chapter.

7. Evaluation

In this chapter we present an evaluation of our methods. In the first section we provide evaluation of the Dependence Discovery method. This evaluation provides understanding of how the various system properties impact the accuracy of the discovered DGs. In the following section, we present evaluation of the Fault Localization method. We examine the precision of the two ranking algorithms and show how the accuracy of DGs impacts the precision of the ranking algorithms. In the next two sections, we evaluate the two methods in conjunction with the Distributed Data Harvesting method applied in environment with limited reachability between network nodes. Hence, in each section we built upon findings in the previous sections. At the end of each section we provide summary of the main finding of the section.

7.1. Evaluation of Dependence Discovery

In our dependence discovery method, dependence graphs are constructed on demand by a discovery element. The graphs are rooted at a given client, beginning at a given time instant, and for some time window. The data provided to the discovery element include both relevant and irrelevant information, since any given monitor will provide data about *all* interactions traversing its associated service during the time window. These interactions involve not just those of the given conversation of interest, but also those of others.

Under such circumstances it would be difficult for a dependence discovery method to provide a perfect result. Moreover, the method by design loses information (e.g., monitors retain only aggregate data, not individual messages) and is sensitive to the dynamics of the operational environment.

Thus, the evaluation questions of interest center mainly on the accuracy of the resulting dependence graph. In particular, we examine the following factors that should influence accuracy in this setting and we look at few additional aspects of our method:

1. **Time window size:** The size of the time window is the main parameter of the

method. With a short time window we may not observe some critical dependencies and with a long time window we may include obsolete and irrelevant dependencies. We explore the impact of the size of the time window on the ratios of false positive and true positive dependencies in the discovered DG.

2. **Degree of service connectivity:** We examine the impact of the degree of service connectivity, which represents the complexity of a service-based system. The higher the degree of interconnection between the services, the larger the number of services and overlap among conversations, and the more noise in the dependence data.

3. **Dynamics of service interconnection:** We examine the sensitivity of the method to a range of rates at which service dependencies change as services adapt to the network dynamics. We would expect that the higher the rate of change of the service dependencies, the more irrelevant dependencies will be included in the discovered dependence graph.

4. **Client workload rate:** We examine the sensitivity of the method to a range of rates at which clients issue service requests. We would expect that as the rate increases, the higher the overlap among conversations, and the more noise in the dependence data.

5. **Mobility speed:** We examine the sensitivity of the method to a range of mobility speeds of the network nodes. We would expect that as the mobility speed increases, the method might have difficulty maintaining consistent results.

6. **Inter vs. intra dependence:** Throughout the experiments, we explore whether the intra-dependence based discovery produces more accurate results when compared to the inter-dependence based discovery.

7. **Comparison with other method:** We compare our method to other methods.

8. **Data storage and transfer requirements:** We examine the data storage and data transfer requirements of our method.

Notice that the degree of service connectivity, the dynamics of service interconnections and the client workload rate are application properties, the mobility speed is an operating environment property, and the time window size is a tuning parameter for the method.

In our experiments we envision military collaborative application providing complex service environment to support mission activities. We consider an army unit of 50 personnel which operates in densely populated area among buildings either on the ground or in vehicles. Each member is equipped with mobile computer device providing mission applications which are consuming range of services hosted either locally or on other devices in the network. The service topologies range from simple services accumulating information from all clients (e.g.: immediate unit status service) occasionally using some other services to complex services using several other services to complete each client request (e.g.: operational command services) which require services such as map, positioning, status of unit members, and others. Other complex composite services use multiple sensors carried out by members of the unit. Topology of this type of service composition is explored in [27]. Because of the wide range of possible service topologies, we explore various degree of service connectivity (i.e. complexity of service interconnections) later in detail in ours experiments.

7.1.1. Evaluation metrics and parameters

Our experiments focus on a particular hypothetical conversation C. A good result for our method would be that it can discover as many dependencies of C as possible, while not including the dependencies of other conversations. We use two metrics to characterize the quality of our results, namely the ratio of true positives (TP) and the ratio of false positives (FP), defined as follows:

$$TP\ ratio = \frac{|D(C) \bigcap GT(C)|}{|GT(C)|}$$

$$FP\ ratio = \frac{|D(C) - GT(C)|}{|D(C)|}$$

where $D(C)$ is the set of discovered dependencies, $GT(C)$ is the set of ground-truth dependencies, true positives are in the intersection of these two sets, and false positives are in the set difference. The TP ratio represents the fraction of discovered dependencies as compared to the actual dependencies in the conversation, and is therefore equivalent to the *recall* metric of information retrieval (taking “ground truth” as “relevance”). The FP ratio represents the fraction of discovered dependencies not belonging to C, and is therefore the complement of *precision.* We assume $D(C)$ and $GT(C)$ are non-empty.

A high TP ratio indicates that most of the dependencies in C have been discovered.

A high FP ratio indicates that the discovery result mistakenly includes a large number of dependencies of conversations other than C. These irrelevant dependencies need to be minimized in order to improve the accuracy of the DG.

We use the service-based system simulator described in the Section 6.1 in the evaluation of the method. Table 7.1 summarizes the basic network-layer parameters used in our simulations. These are standard settings used widely in the networking community and embodied in the NS-3 simulator. Specifically, we use the log distance model with path-loss exponent 3 for wireless signal propagation, reproducing a network operated in an urban area [24]. We set the spatial mobility bounds to a 75 meter square, which is a limitation imposed by the chosen WiFi standard, as larger regions induce long-term network partitioning. Another important parameter is the mobility speed of the mobile hosts. For most of the experiments we set the mobility speed of all nodes to 10 m/s. This is a challenging scenario that allows us to simulate environments in which message exchanges between the nodes are materially affected by the temporary disruptions in communication at the network layer. In Section 7.1.5 we report results in which we vary this parameter from 0 m/s (i.e., a "stationary" wireless network) to 15 m/s.

Table 7.2 summarizes the basic service-level parameters used in our simulations. The message sizes and timeouts are derived from standard values found in SOA and Web Services implementations. Note that the number of methods in each service is not significant from a simulation point of view, as long as we have at least two methods available so that we can examine the impact of intra-dependence information on dependence discovery.

For the interconnection model, we use a 2-tiered topology. The first tier consists of the connections between the clients and a set of "front end" services, while the second tier consists of the connections between the services themselves. In our simulations, we use 50 clients, five front-end services, and 25 "back end" services. When starting a conversation, each client invokes a method selected uniformly at random from all methods provided by the five front-end services. We experimented with other topologies, including a single-tier topology in which there are no designated front-end services, and with different numbers of clients and services, but found that the results were consistent. We therefore only report results based on the 2-tier topology.

Of particular importance is the degree of connectivity among the services. Table 7.3 shows how we configure the simulations to capture three different connectivity scenarios, denoted as "Low", "Medium", and "High". The connectivity degree is induced by the probability that a method in one service invokes a method in another service; the higher the

Number of nodes	50
Spatial bounds	75m x 75m
Mobility speed	10 m/s
Mobility model	RandomDirection2dMobility
Propagation delay model	ConstantSpeedPropagationDelay
Propagation loss model	LogDistancePropagationLoss/α3
WiFi standard	80211b
WiFi rate	11Mbps
Routing protocol	OLSR and Ipv4StaticRouting
Protocol stack	UDP/IPv4

Table 7.1.: Network-layer parameters

Number of clients	50 (one per node)
Number of services	30
Invokable methods per service	2
Workload (client request rate)	10s – 80s
Service dependence switch rate	5 minutes – 15 minutes
Message size	500 bytes – 1500 bytes
Response timeout	60s

Table 7.2.: Service-layer parameters

	Low	Medium	High
Inter-service, method-to-method connectivity probability	0.0125	0.025	0.05

Table 7.3.: Service-connectivity parameter

	Low	Medium	High
Client-to-service dependencies	250	250	250
Service-to-service dependencies	30	59	130
Dependencies in graph (avg.)	2.03	3.21	7.9
Dependencies in graph (std. dev.)	1.2	2.28	3.63

Table 7.4.: Dependencies induced by connectivity

probability, the denser the interconnection topology.

The effect of different connectivities on the resulting ground-truth dependencies for each scenario is shown in Table 7.4. Because the 50 clients will always invoke the five front-end services over the course of an experimental run, there will be 250 client-to-service dependencies in all scenarios. However, the total number of unique service-to-service dependencies increases with the connectivity probability, from 30 dependencies for low connectivity to 130 dependencies for high connectivity. Also increasing with the connectivity probability is the resulting number of dependencies per conversation (i.e., the number of arcs appearing in the ground-truth dependence graph rooted at a given client). The average and standard deviation for the conversations occurring in the three scenarios are given in the table. Notice that a high standard deviation in this case indicates a good mix of different kinds of conversations.

An artifact of having more services involved in a conversation due to a higher degree of connectivity is that it increases the time it takes to complete the conversation. This is borne out by Figure 7.1, which gives the cumulative distribution of the conversation lengths in our simulations for the three connectivity scenarios. For example, 80% of the conversations in the low connectivity scenario are completed in under 0.2 seconds, whereas the top 20% of the conversations in the high connectivity scenario take longer than 4 seconds.

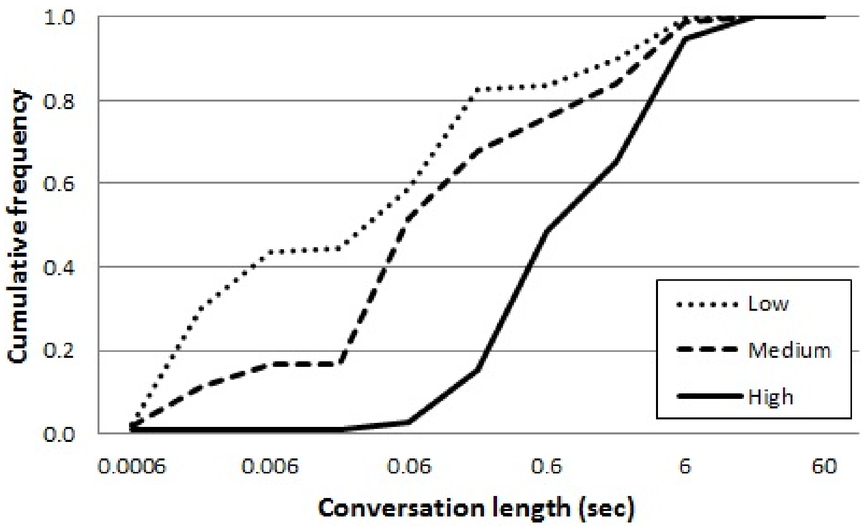


Figure 7.1.: Cumulative distribution of conversation lengths (based on request messages) for three connectivity scenarios

Service dependencies may change not only due to normal computational progress, but also to optimize the use of otherwise equivalent services. In this way the service system adapts to the network dynamics, such as large fluctuation in communication quality and the availability. The services may be categorized into a stateless or stateful according to the model of their internal state management. The stateless services do not maintain any internal state between invocations requested either by a same client or by different clients. This model allows the services to select antecedent services needed for completion of invocations, based on current availability rather than on state or history. The stateful services are more likely to use previously used antecedent services already holding state arising from previous invocations. Thus, the stateless model leads to higher dynamicity in the servicesŠ bindings and dependencies than the stateful model. To see the impact of this form of dynamic service rebinding, we introduce into our experiments a *service-switching* behavior. Every service is given two designated alternative services, and every dependent service then switches among these three services after a randomly selected time period. In Section 7.1.3 we report results in which we vary the switching-time parameter in increments between 10 seconds on average (representing an extremely dynamic behavior) and infinite (representing a non-switching behavior).

We collect our results from 30 minutes of simulated execution time after excluding 30 seconds of warm up. Each combination of parameters in our experiments results in thousands of conversations occurring during the simulated 30-minute execution. For instance, the low, medium, and high connectivites combined with 10-minute switching-time periods and 10 m/s mobility speeds result in 8440, 8139, and 7165 conversations, respectively. The results given below are averages over the data collected from these conversations, where each conversation is then a statistical sample subject to the random variables.

7.1.2. Impact of time window size and service connectivity

We first look at the impact of the time window size and connectivity degree on the accuracy of the results. We hypothesize that as the time window size grows, so too should the TP ratio, since more dependencies will be captured. However, increasing the time window size should also increase the FP ratio, since there is a greater chance that messages belonging to other conversations are included in the dependence graph. For a given time window size, we expect the TP ratio to be negatively correlated with the connectivity degree, since a higher connectivity increases the conversation length, which in turn increases the chances that some dependencies are missed. Similarly, we would expect the FP ratio to be higher in densely connected service configurations, since dependencies in other conversations are more likely to overlap those of the subject conversation.

We calculate the TP and FP ratios for both inter- and intra-dependence discovery separately. Figures 7.2, 7.3 and 7.4 depict the results, where each data point is the ratio averaged over all conversations. Here we consider only fixed service bindings (i.e., non-switching behavior). The variances of the TP and FP ratios are small, and therefore omitted from the figures. For example, the largest 95-percentile confidence intervals for TP and FP ratios in the medium connectivity scenario are 0.006 and 0.0053, respectively.

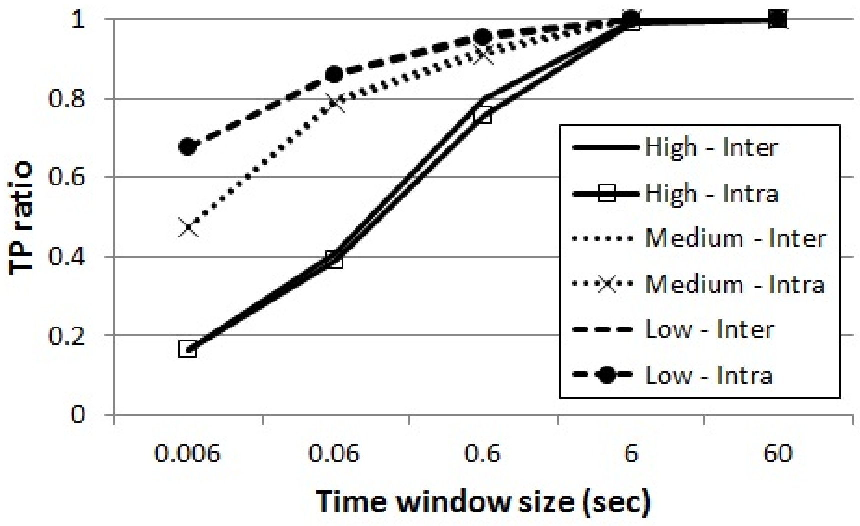


Figure 7.2.: Accuracy of inter- and intra-dependence discovery methods for different time window sizes given as TP ratios

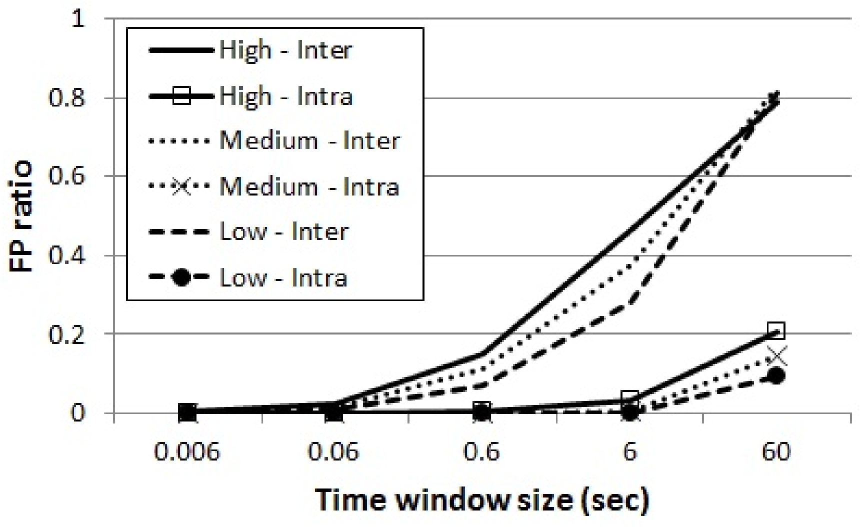


Figure 7.3.: Accuracy of inter- and intra-dependence discovery methods for different time window sizes given as FP ratios

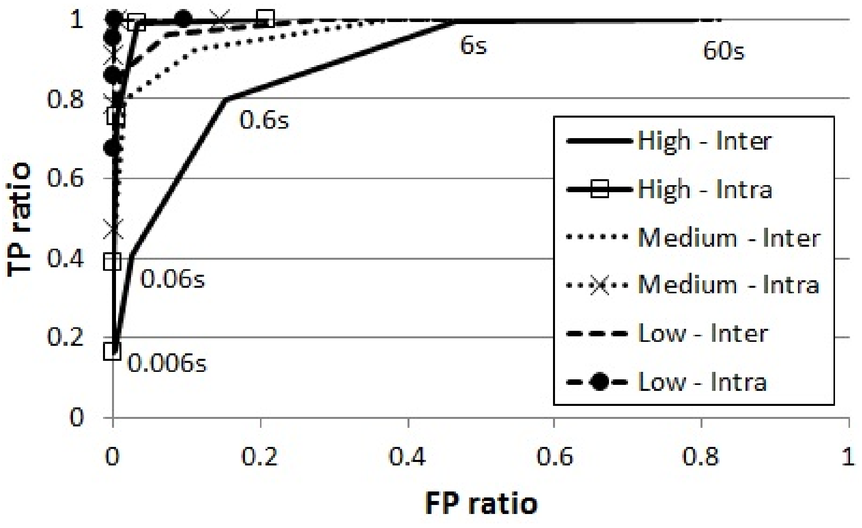


Figure 7.4.: Accuracy of inter- and intra-dependence discovery methods for different time window sizes given as a trade off between TP and FP ratios

As shown in Figure 7.2, increasing the time window size increases the TP ratio, both for inter- and intra-dependencies. A larger window will include more messages and thus discover more dependencies. However, increasing the time window size also increases the FP ratio, as shown in Figure 7.3. A larger window will include more messages from other conversations. The same figures also confirm our hypotheses about the impact of the connectivity degree: the TP ratio decreases and the FP ratio increases as the service topology becomes denser. Notice, too, that intra-dependence discovery has a significantly lower FP ratio than inter-dependence discovery. This is due to the fact that it can precisely correlate incoming and outgoing inter-dependencies, something to which inter-dependence discovery is blind (recall Figure 3.2). To directly display the tradeoff between the TP and FP ratios under various time window sizes, we plot them against each other in Figure 7.4, where each point represents the given time window size.

7.1.3. Impact of dynamic service rebinding

We now investigate the sensitivity of the method to dynamic service rebinding (i.e. the dynamics of service interconnections). For a given time window size, we would expect a high switching rate to cause many dependencies belonging to irrelevant conversations to be included in the discovered dependence graph, thereby increasing the FP ratio. In contrast, we would expect the TP ratio to be insensitive to the switching rate because once a message is seen, any additional messages, relevant or irrelevant, should not increase that ratio.

We ran experiments using seven different average switching-time periods, from 10 seconds up to a case in which no switching occurs; a lower value results in a faster switching rate. Figure 7.5 shows the intra-dependence FP ratio in the medium connectivity scenario. (As hypothesized, the TP ratio is essentially unaffected by the switching rate, so we do not show that result here.) The effect on FP ratio is most noticeable when the switching-time period is on the same order as the time window size, which indicates possibly many service switches happen within the time window, rendering the resulting dependencies irrelevant from the discovery element's point of view.

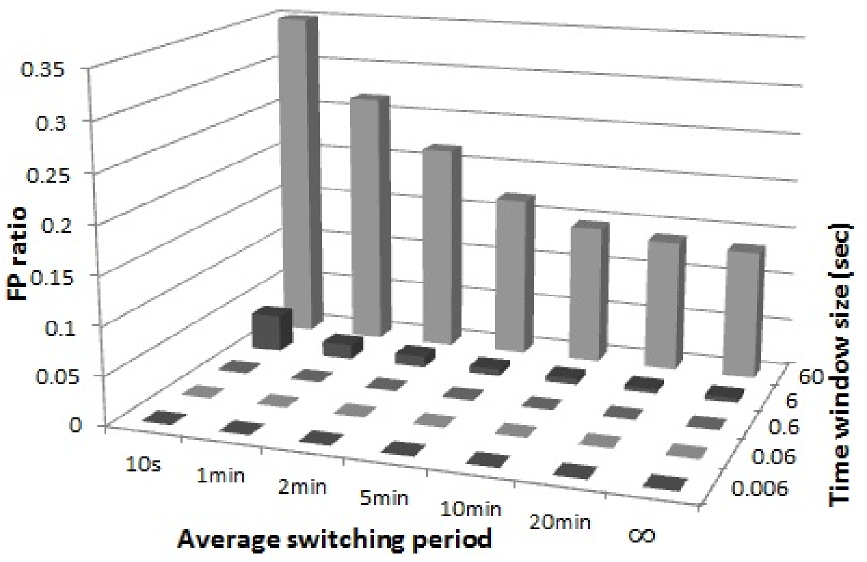


Figure 7.5.: FP ratio for intra-dependence discovery under medium connectivity for various switching-time periods and time window sizes

7.1.4. Impact of the client workload

We now investigate the impact of the client workload on the TP and FP ratios. We define the workload to be the rate at which clients issue service requests. We hypothesize that the workload has a positive correlation with the FP ratio, since a high workload will generate more messages that are part of irrelevant conversations. In contrast, we expect the TP ratio to be insensitive to the workload. This is because the TP ratio is related to the fact of messages being exchanged between services, rather than to the volume of those messages. In other words, once a message is seen to have been exchanged between two services, any additional messages, relevant or irrelevant, should not increase the TP ratio.

Using the high dependence scenario, we experiment with four different workload rates: 10s, 20s, 40s, and 80s. Each rate represents the average of a uniformly random waiting time from the completion of a given service request by a client to the issuance of the next request by that client. A lower value therefore indicates a higher workload.

Figures 7.6, 7.7 and 7.8 present our results. Overall, the effect of the workload appears to be minor, with only slight differences in TP and FP ratios evident as the length of the time window increases. This is what we expected to see for the TP ratio, but the very weak correlation for the FP ratio is a somewhat surprising result. We hypothesize that this is an artifact of the particular service configuration used in the experiments

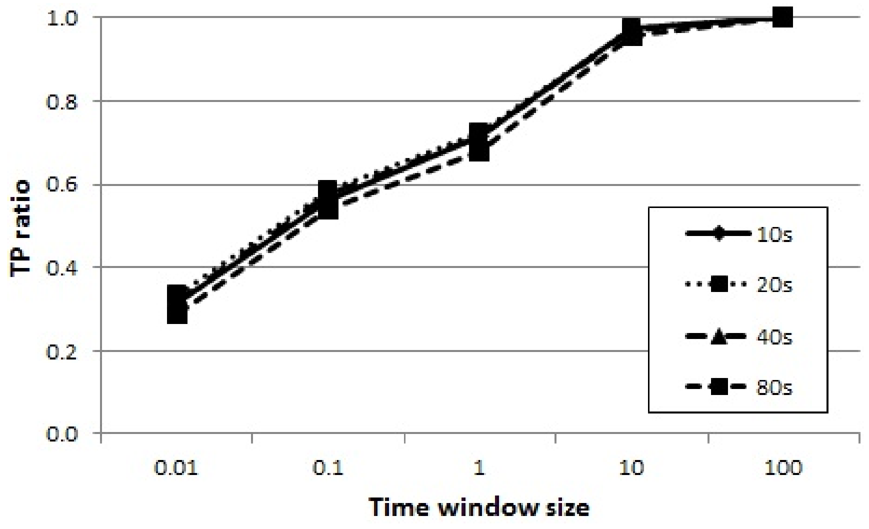


Figure 7.6.: TP ratios for inter-dependence discovery under four different workloads. As the time window length increases, the TP ratio differences are negligible

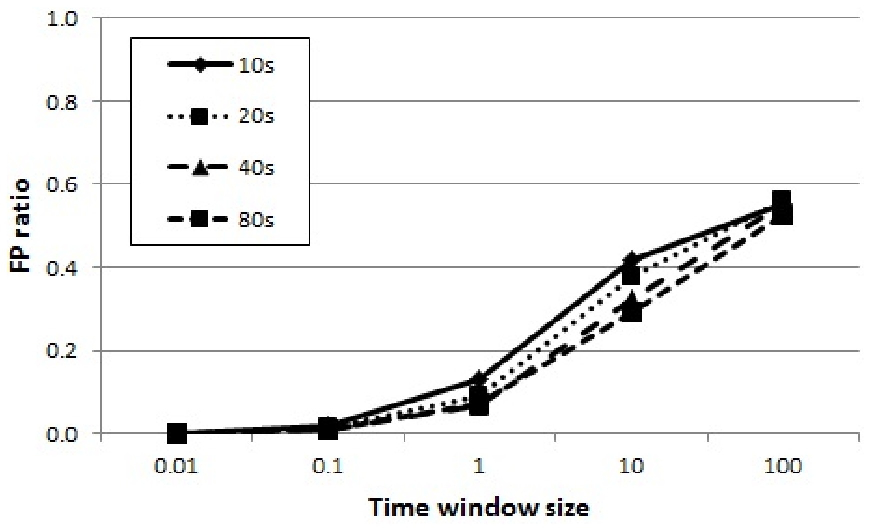


Figure 7.7.: FP ratios for inter-dependence discovery under four different workloads. As the time window length increases, the FP ratio differences are small

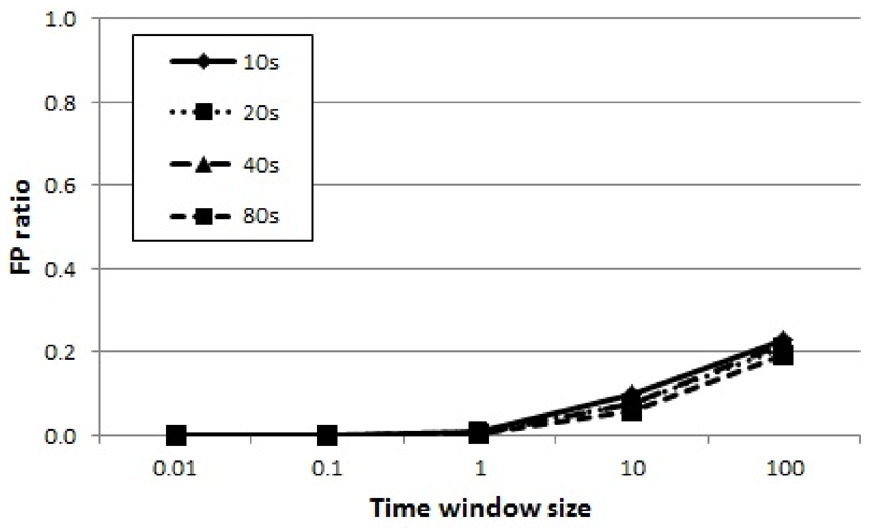


Figure 7.8.: FP ratios for intra-dependence discovery under four different workloads. As the time window length increases, the FP ratio differences are small

7.1.5. Impact of the mobility speed

The next set of experiments investigate the impact of the mobility speed on the inter-dependence discovery TP and FP ratios. We work with four different mobility speeds: 0 m/s (amounting to a fixed network), 5 m/s, 10 m/s, and 15 m/s.

We would expect that as the speed of the mobile hosts increases, the quality of the links between them should deteriorate and, consequently, the failure rate of message exchanges should increase. Higher message failure rates should in turn increase conversation lengths, as more messages must be resent in an attempt to complete the conversations. Therefore, increasing the mobility speed should have the effect of decreasing the TP ratio, since messages require longer time periods to be exchanged and detected. This, however, should not impact the FP ratio, since the proportions of relevant and irrelevant messages remain unaffected.

The results are reported in Figures 7.9, 7.10 and 7.11. As expected, the TP ratio generally has a positive correlation with mobility speed, with the fixed network (0 m/s) exhibiting the highest ratio. Furthermore, we observe that the lengths of the conversations increase with mobility speed (and network failure rate), which is visible in the TP ratios between the 0.006s to 0.6s time windows. However, for longer conversations, the time windows of 6s and 60s are long enough to capture all cases. The FP ratios are virtually unaffected. This coincides with our understanding that the ratio of relevant and irrelevant messages remains the same.

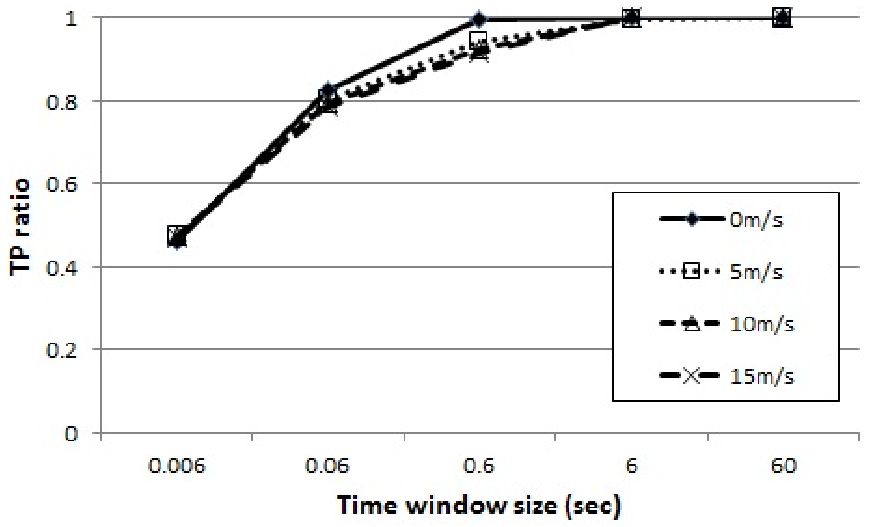


Figure 7.9.: TP ratios for inter-dependence discovery under four mobility speeds

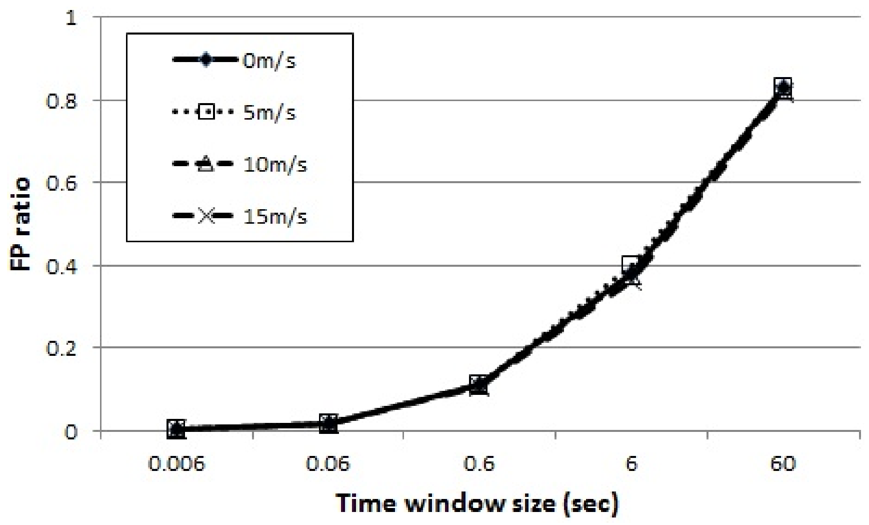


Figure 7.10.: FP ratios for inter-dependence discovery under four mobility speeds

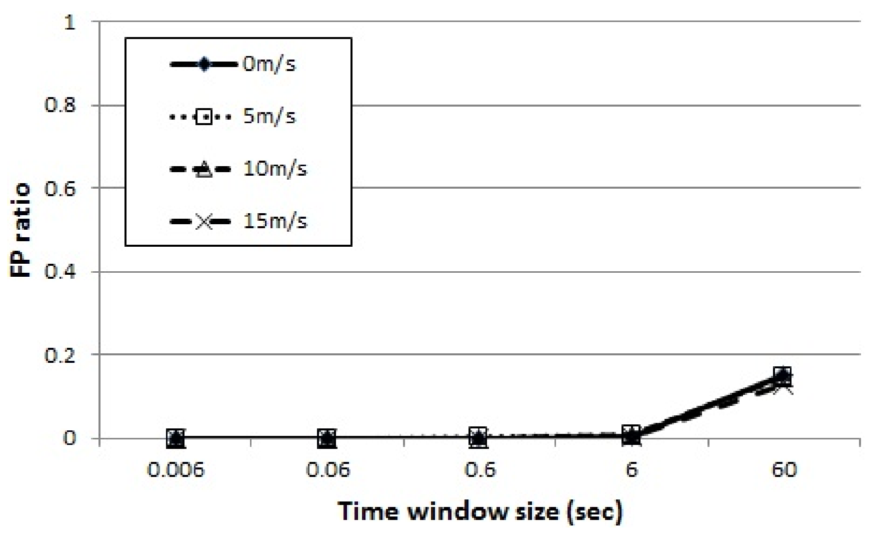


Figure 7.11.: FP ratios for intra-dependence discovery under four mobility speeds

7.1.6. Comparison with existing methods

We now compare the accuracy of our dependence discovery method to that of existing methods. We make this comparison by implementing two alternative methods to represent the two major classes of existing approaches: those that perform discovery at the service level [9, 16] and those that perform discovery at the network level [5, 6, 7, 8, 19, 43]. These implementations are, like our own method, prototyped as queries over the trace database (see Section 6). The service-level alternative discovers a global system dependence graph by observing all the service messages exchanged over the whole execution period and, from this, builds dependence graphs for the individual client conversations. The network-level alternative works similarly, but only observes the flow of messages by inspecting the information contained in the headers of packets exchanged over the relevant IP ports. It then builds dependence graphs using external information provided to it about the deployment of clients and services on hosts.

We compare our method against the two alternatives using the medium connectivity scenario, a 10 minute service-switching time period, and a 10 m/s mobility speed. The average number of ground-truth dependencies is 3.21 (see Table 7.4). We configure our method to use a 60s time window size, which is large enough to capture all such dependencies (see Figure 7.1). The comparison then reduces to one based on the false dependencies appearing in the discovered dependence graphs.

As we discuss in Section 2, the existing service- and network-level methods are designed for use in fixed networks and for relatively stable service configurations. Therefore, since both these alternative methods build dependence graphs from long-term observations, we do not expect them to adequately filter out stale dependencies caused by the dynamics of the scenarios, resulting in higher false positives than with our discovery method. Moreover, the network-level method should include even more false positives than the service-level method because it builds dependence graphs from coarser-grained information.

The results are reported in Figure 7.12, where a vertical line is used to separate the results for our method on the left from the results for the alternative methods on the right. We give the FP ratio, as well as a count of the false dependencies appearing in the dependence graph. Both are computed as the average over the total number of conversations (8139) occurring in the 30-minute execution period. As expected, our method provides dependence graphs having a significantly lower FP ratio than the alternative methods. Furthermore, although the FP ratio for our inter-dependence discovery is similar to that of the alternative methods, the actual number of false dependencies is significantly lower.

Note that FP dependencies is number of false dependencies in resultant graph whereas FP ratio is the proportion of FP dependencies in the resultant graph as defined in section 7.3.1.

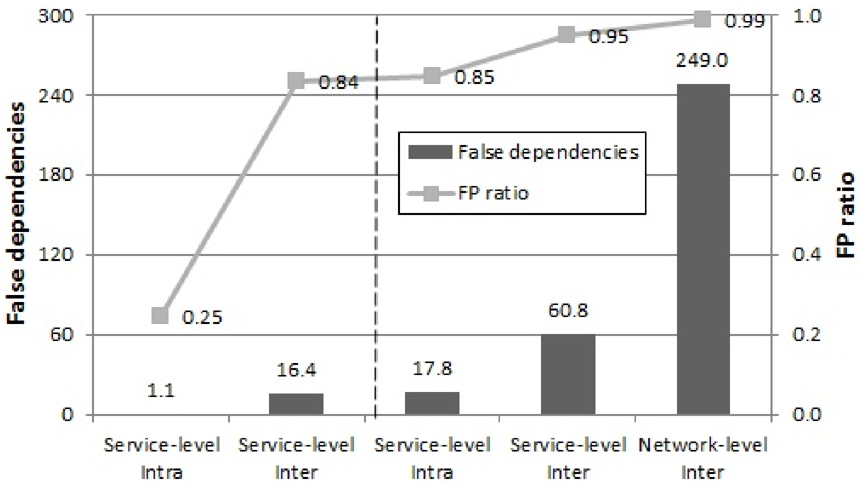


Figure 7.12.: Comparison of methods in medium connectivity scenario. Results for existing methods are to right of dashed line

7.1.7. Estimates of data storage and data transfer needs

We conclude our evaluation of the dependence discovery method by investigating the data storage requirements placed on the monitors, and the data transfer requirements placed on the network. We do this by positing a representation for the data, analyzing the space requirements of that representation, and computing the total for a particular experimental scenario.

Consider the medium connectivity scenario, where the average number of inter- and intra-dependencies per service is 12 and 53, respectively. Assume a maximum size for identifiers of 256 single-byte characters, which corresponds to the Web Services framework's use of the maximum URL length. With a time slot length of 0.01 seconds (an especially precise configuration) maintained for a time period of 10 minutes, using formula from Section 3.2 the required data storage will amount to approximately 506 kilobytes per monitor.

The estimates above are in some sense worst case. In practice the identifiers will be smaller (certainly smaller than the maximum allowed URL length). Moreover, the storage used for intra-dependence identifiers can be significantly reduced, simply by maintaining them as references to the corresponding incoming and outgoing inter-dependencies.

While the monitors locally store data describing all dependencies detected during the time period, the data transferred over the network to the discovery element contains only the data for the desired time window. This time window is typically much shorter than the stored time period.

Consider again the medium connectivity scenario. The typical data bundle transferred by an individual monitor contains data for approximately one outgoing inter-dependence and one intra-dependence. This amounts to a maximum of 512 bytes of data per service. The average number of services involved in a client request is 3.21, so the discovery element will issue on average 4.21 requests (including the initial request to the client's monitor) for dependence data in order to construct the dependence graph. This results in a total of approximately 1.6 kilobytes of data transferred over the network, which is substantially less than the total amount of dependence data stored by the monitors.

7.1.8. Discussion of the results

The experiments presented above establish how the time window size, degree of service connectivity, dynamics of service interconnection, client workload rate and mobility speed affect the accuracy of our dependence discovery method. Connectivity, interconnection dynamics, workload and mobility speed are properties of the application and network, while time window size is an operator's tuning parameter. This represents a broad exploration of the behavioral characteristics of our method.

In addition to the results reported here, we have experimented with several other parameters such as rates of service faults. However, these parameters do not seem to have a significant impact on the accuracy of the method.

In general, the results for intra- and inter-dependence discovery indicate consistently equivalent accuracy as measured in terms of TP ratio. On the other hand, the FP ratio is significantly lower for intra-dependence discovery compared to that for inter-dependence discovery. Both the TP and FP ratios are significantly affected by the selection of the time window size: as the time window size increases, so do the TP and FP ratios. The TP ratio, however, tends to reach a maximum value at a certain time window size, whereas the FP ratio degrades (i.e., increases) unabated beyond that value. This implies that the time window can be selected such that it will yield a discovery result striking a good balance between true positives and false positives.

We can also see that the FP ratio is significantly affected by the service-switching time period. This factor depends on the service discovery and selection technique used in the system. In general, however, as TP ratio is relatively insensitive to the service interconnection dynamics, one can select the time window size after taking into account the service switching rate, which in a MANET is typically related to the dynamics of the network.

In contrast to time window size, the accuracy of the dependence results appears to be less sensitive to network mobility and workload factors, assuming the service interconnection dynamics are at a given level. This implies that our method is relatively robust to variations in operational conditions.

The comparison with existing methods validates our hypothesis that MANET-hosted service-based systems require a fundamentally different approach to dependence discovery. Particularly striking is the difference in FP ratio between our method's intra-dependence and the other methods. While our method provides approximately one false dependence on average per dependence graph, the existing approaches yield dependence graphs significantly larger and constructed mostly from false dependencies. This is especially evident in the network-level approach, which yields dependence graphs containing almost whole the entire set of services, rendering dependence discovery virtually useless.

Finally, our estimates for the method's storage requirements fall well within the capabilities of today's mobile hosts. Moreover, the method's aggregation of data results in an extremely low requirement for data transport.

7.2. Evaluation of Fault Localization

In the evaluation of the fault localization method, the questions of interest center mainly on the accuracy of the resulting candidate set, specifically the ranking of root-cause candidates. A good result, of course, would be that our method consistently ranks the correct root-cause candidate at or near the top. We examine the following factors that might influence accuracy:

1. **Ranking algorithm:** We compare the two ranking algorithms, timing and BNet, in terms of the position at which they place the correct root-cause failure.
2. **Dependence graph accuracy:** The dependence graph (DG) that results from the dependence discovery mechanism may contain some services that are irrelevant to the conversation of interest, causing the FPM induced from the DG to include failure symptoms irrelevant to the failure observed by the client. We examine the effects of DG accuracy—more precisely, the ratio of false positives—used in the construction of an FPM.
3. **Service connectivity:** We examine the impact of service connectivity, which represents the complexity of a service-based system. The higher the degree of interconnection between the services, the larger the number of services and overlap among conversations, and the more noise in the dependence and symptom data.
4. **Service fault rate:** We examine the sensitivity of the ranking algorithms to a range of service fault rates. We would expect that as the rate increases, the algorithms might have difficulty maintaining consistent results.
5. **Data storage and transfer requirements:** We examine the data storage and data transfer requirements of our method.

7.2.1. Methodology and evaluation criteria

In the evaluation experiments we use similar setup of the Service-based system simulator as in the evaluation of the dependence discovery method. For the network-level behaviors, such as wireless signal propagation models, we use standard settings used widely in the networking community and embodied in the NS-3 simulator. We configure a network of 50 hosts, and simulate the random mobility of the hosts at a fairly high speed (10 m/s). The significance of high mobility is that the hosts are frequently disconnected from each other and, as a result, the message exchanges between the services fail frequently, causing message send failures (SENDF) and timeout failures (RC_TO) in the context of our fault models.

The service-level parameters used in our simulations are derived from standard values found in SOA and Web Services implementations. We generate a system of 50 clients and 30 services, with the services arranged into five "front end" (client facing) and 25 "back end" services. In addition to the network-level faults, we also cause two kinds of service-level faults: In the first kind, the service fails to generate proper responses to the service requests and instead generates exceptions, while in the second, the service itself fails due to, for example, a host failure, and does not generate any response. In our context, a failure of the first kind corresponds to the root cause of an EX failure, while the second results in the root cause of a TO failure. All but the last of our experiments use a service fault rate of 0.5%, which means 5 out of every 1000 service requests fail. We configure the simulations such 90% constitute exception root-cause faults (SENDF or SF) and 10% timeout root-cause faults (RC_TO).

We collect our results from 30 minutes of simulated execution time after excluding a 30 second warm-up period. Each combination of parameters results in thousands of conversations occurring during the simulation, among which the failed conversations constitute the statistical samples of our analysis subject to the randomness in the service request times, host mobility, wireless signal fading, and the like.

Table 7.5 summarizes the failure symptoms observed in our simulations, where the columns "Low" and "High" represent sparse and dense service connectivity scenarios, respectively. For instance, the low connectivity configuration (combined with 0.5% service fault rate and 10 m/s mobility speed) results in 8541 conversations. The average number of services in the dependence graph of each conversation is larger in highly connected service configurations, as is the ratio of the failed conversations: about 21% of all conversations fail in the high connectivity scenario, compared to about 5% in the low connectivity scenario. Of all root-cause failures, SENDFs constitute about 75%. These are caused by faulty network links, rendering services unable to successfully send request messages to other services. In comparison, timeout failures occupy a small portion of only about 7%. The rest of root-cause failures, about 16% are caused by the injected software failures.

	Low	High
Services in dependence graph (avg.)	2.03	7.9
Services in dependence graph (std. dev.)	1.2	3.63
Number of conversations in scenario	8541	6810
Ratio of failed conversations	5.01%	20.88%
Proportion of SF root causes	16.6%	16.0%
Proportion of SENDF root causes	75.7%	76.8%
Proportion of TO root causes	7.7%	7.2%
Root cause symptoms in timewindow per client fault EX (avg.)	34.88	102.66
Transitive symptoms in timewindow per client fault EX (avg.)	13.41	125.69
Symptoms in timewindow per client TO failure (avg.)	2	8.36

Table 7.5.: Dependencies and faults induced by connectivity

A secondary effect of the ratio of failed conversations is the number of faults occurring in given time period. For example in Table 7.5, the average number of faults occurring in all services of the scenario in relevant time window prior client exception is 34.88 root-cause faults and 13.41 transitive faults, whereas, in high connectivity scenario the number increases to 102.66 root-cause faults and 125.69 transitive faults.

The increasing number of faults in time window increases number of irrelevant symptoms involved in fault localization of given conversation. Irrelevant symptoms are eliminated from analysis with use of DG which limits the number of services from which symptoms are included in analysis. However, with increasing connectivity is increasing size of DG and thus the increasing connectivity has combined negative impact on fault localization due to increasing rate of faults as well as due to increasing portion of system from which the faults are included in analysis. In Section 7.2.4 we report results in which we vary the connectivity parameter.

The impact of having more services involved in a conversation is the increasing time of propagation of faults from root-causes to clients. This is borne out by Figures 7.13 and 7.14, which gives the cumulative distribution of the period between root-cause occurrence and client receiving exception and period between client failing due to timeout and occurrence of root-cause timeout. For example, 92% of the exception faults in the low connectivity scenario propagates in under 0.03 second, whereas only about 50% in the high connectivity scenario.

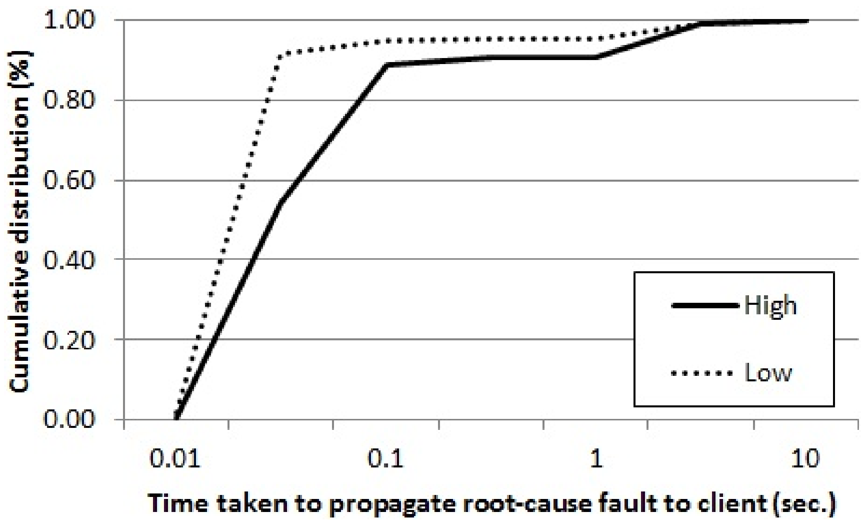


Figure 7.13.: Cumulative distribution of time between occurrence of root-cause faults and when clients witness exception messages

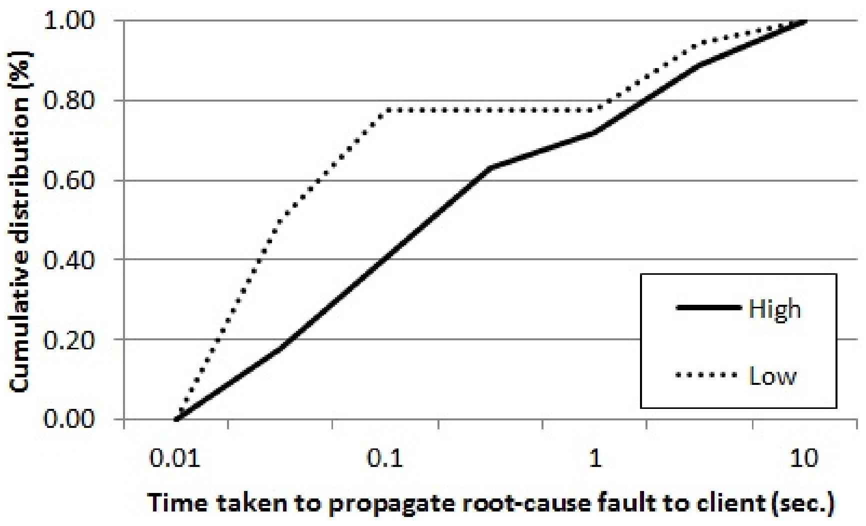


Figure 7.14.: Cumulative distribution of time between occurrence of root-cause faults and when clients witness timeouts

The ratio of fault types in scenario is an outcome of several system properties such as the behavior of the network, configuration of services and the characteristics of message exchanges. When link between services is broken usually request message fails to be sent and the resultant fault will propagate as an exception back to client. However, in some cases, the quality of the link deteriorates after request was sent and later response message fails to arrive back which causes timeout fault. Aside of these network induced faults, services may fail due to software or data problems. Hence, in order to simulate this behavior, we insert software faults into the services such that every service invocation has certain probability to fail drawn from configured probability distribution.

Similar to network, service can fail and respond with exception message which is a common reaction to service fault handled either by the service itself or by the host environment of the service. However, in some cases the service may fail without sending any response exception. This is particularly common fault in MANETs where nodes have limited reliability (i.e. due to limited battery time). We model this fault behavior such that 10% of service faults does not produce response message and thus causes timeout fault. Table 7.5 contains resultant ratios of root-cause faults of failed conversations, in scenarios with service fault rate 0.5%. In all scenarios most of the faults are caused by inability of services to send request message to another service (SENDF) i.e. over 70%. Send faults along with service faults (SF) propagate as exceptions and thus over 90% of all faults in all scenarios cause clients to fail with exception. We have experimented with various ratios of the SENDF and SF root-cause faults; however, the ratios have no impact on precision of the fault localization. Timeout faults represent only about 7% of all client faults. This is a significant observation because the localization of timeout faults is less precise (due to root cause as well as transitive symptoms being indistinguishable). However, because the fraction of timeouts is relatively small it has thus small negative impact on overall precision of the fault localization.

Note, that the various system variables (i.e. mobility speed, Clients workload or dynamic dependencies) impact the fault localization as a variation of the ratio of FP dependencies in DG. We have explored the impact of these variables on FP ratio in the evaluation of the dependence discovery method in section 7.1. In this section we are concerned with the overall impact of the FP ratio on the fault localization regardless of origin of the false dependencies and thus we do not explore impact of each individual variable independently.

Selection of time window size

As mentioned in Section 4.3.2, the time window is an important tuning parameter for our method, as it affects which failure symptoms are included in the FPM. When selecting the size of the time window, one needs to consider how long it would take for root-cause faults to propagate to clients. A naïve approach would be to set the value equal to the longest duration that a conversation can last, which in turn is limited by the service response timeout parameter, since any failure in the system should occur within this period. Following standard .NET and similar implementations, the timeout parameter is set to 60 seconds in our experiments. In practice, however, the failure propagation time can be much shorter than the duration of the entire conversation, and hence the time window size also needs to be selected accordingly.

Figures 7.13 and 7.14 give the cumulative distribution of the propagation time for exception and timeout root-cause faults to reach clients in each of the two service connectivity scenarios. We can see from the distributions, regardless of the kind of failure and service connectivity, that the time difference between the failure seen by the client and the root-cause failure is fairly small: in almost all cases, the failures are propagated within a few seconds. This result suggests that the time window size indeed needs to be set much smaller than the maximum duration for the conversation so as not to include too many irrelevant symptoms in the analysis. In the results below, we generally use a 6-second time window for this purpose. Note, however, this value is chosen based on empirical results, and hence should be considered as one that is close to ideal choice. To reflect the cases when such an empirical basis is not available, results with the 60-second time window are also shown whenever applicable.

7.2.2. Comparison of ranking algorithms

We first compare the accuracy of the two ranking algorithms. To isolate their effect from those of other parameters, we show the results for high service connectivity with 0.5% service fault rates, using 100% accurate dependence graphs (referred to as "ground truth" in Figure 6.13), which contain exactly the services involved in each conversation.

Accuracy is measured in terms of the ranking position of the correct root-cause fault for each failed conversation in the experiment. The results are given in Figures 7.15 and 7.16, which shows the cumulative distribution of those positions. For purposes of comparison, we include the results for the BNet-based algorithm under both 6- and 60-second time windows; The results by the timing-based algorithm are virtually the same for both time window sizes. Notice that the average sizes of the candidate sets under the 6-second time window are 1.5 for exceptions and 2.6 for timeouts, and 5 and 3.5, respectively, under the 60-second window.

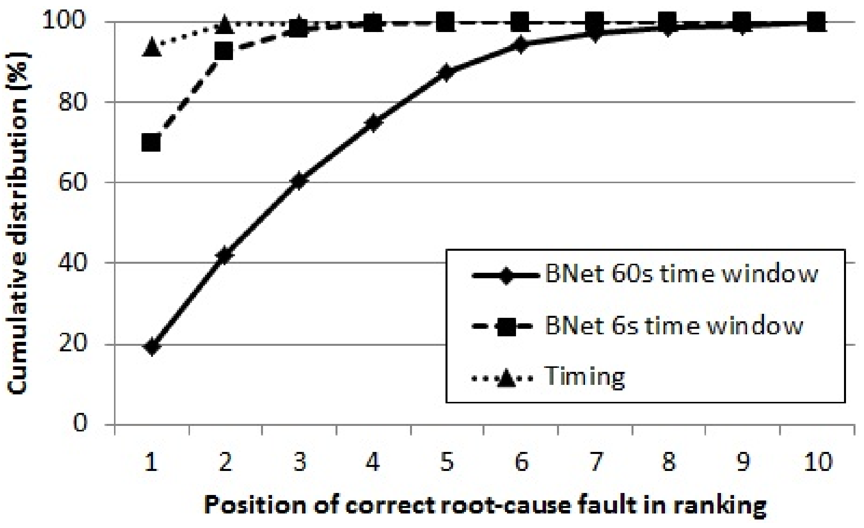


Figure 7.15.: Accuracy of ranking algorithms for exception faults in high connectivity scenario. The FPM is constructed from the ground-truth dependence graph

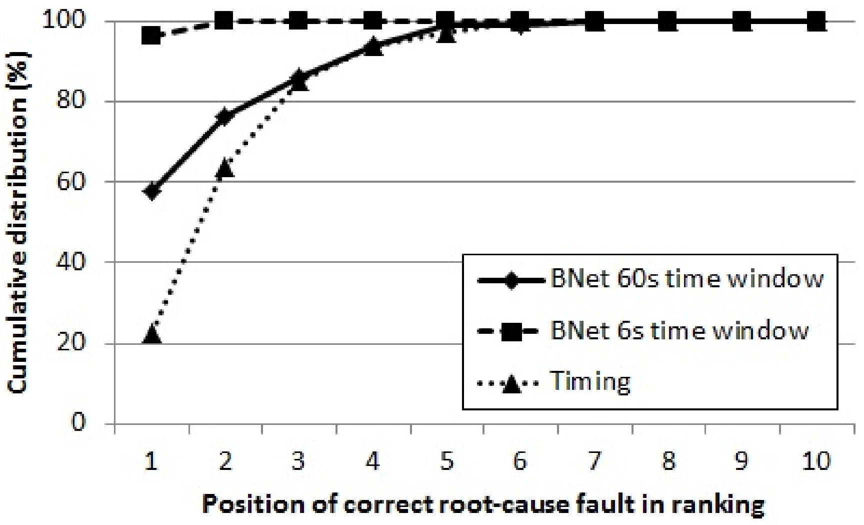


Figure 7.16.: Accuracy of ranking algorithms for timeout faults in high connectivity scenario. The FPM is constructed from the ground-truth dependence graph

We can see that the timing-based method is very effective for exception failures, resulting in more than 90% of correct root causes placed in the first position of the ranking, under both 6-second and 60-second time windows. Compare this with an ordering of candidates by random guess, which would do so only for 66% and 20% under 6- and 60-second time windows, respectively. This good result is due to the fact that exception failures are propagated through a quick succession of explicit messages, which makes a ranking based on the time proximity particularly suitable for such failures.

On the other hand, the BNet-based algorithm is more effective in ranking the root causes of timeout failures. Under a 6-second time window size, the BNet algorithm places the correct root causes in the first position for more than 95% of the failures, and for nearly 60% under 60-second time window. Compare this again with a random ordering, which would achieve the same only for 38% and 28% of the cases under the two time window sizes, respectively. The accuracy of the timing-based algorithm is not nearly so good for the timeout failures.

The results also show the importance of the time window size for the BNet algorithm. Using the extreme case of 60 seconds, the algorithm produces relatively poor rankings. This is because the data include a substantial amount of noise: extraneous symptoms, irrelevant to the failure witnessed by the client. In fact, the candidate sets are much larger, with on average 10 timeout and 19 exception candidates, about three times more than for a 6-second time window in these experiments. Nevertheless, BNet-ranking performs much better by placing the correct root cause at the first position for about 60% of timeouts; compare it with the results by a random guess, which would do so only for 28% out of 3.5 candidates.

7.2.3. Impact of dependence graph accuracy

We next investigate the sensitivity of our method to the accuracy of the dependence graph (DG) rooted at a client. Recall that the DG is created using a run-time discovery facility that produces probabilistically correct results. Its accuracy can be measured in terms of any extraneous dependencies included in the graph. An extra dependence is a false positive (FP) drawn from some other, irrelevant conversation. The ratio of false positives to true positives (the FP ratio) is influenced by various factors, such as mobility speed, service workload, and most significantly the particular technique used for dependence discovery.

Here we make use of the intra- and inter-dependence discovery techniques. They are represented here, respectively, by the FP ratios 3% and 46% in our experiments. We also include in our experiments the ground-truth DG that contains no false positives. Again, we show results for the high connectivity scenario and 6-second time window.

Figure 7.17 shows the impact of the FP ratio on the accuracy of our method under 6-second time window. The average ranking positions of correct root causes for exception and timeout faults are plotted against each other and shown for the two ranking algorithms.

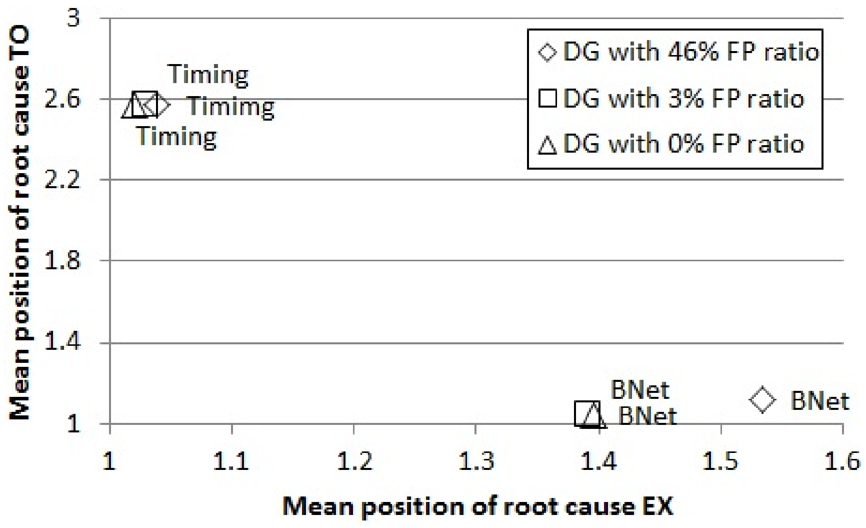


Figure 7.17.: Impact of dependence graph false positive ratio on mean ranking position of correct root-cause fault. Ratio of 0% represents the ground truth.

Consistent with the results in Section 7.2.2, the algorithms behave differently for the two kinds of faults. Notice that the timing-based algorithm is essentially insensitive to the accuracy of the DG, which reinforces our conclusion that its effectiveness at localizing exception faults is dominated by the time proximity of the fault occurrence. The BNet algorithm also exhibits only some small sensitivity to the DG's accuracy.

7.2.4. Impact of service connectivity

We now turn to the impact of service connectivity (i.e., system complexity) on the accuracy of our method. It is important to investigate this factor because it influences many other contributing factors, such as the size and false positive ratios of dependence graphs, the duration of conversations (and, consequently, the amount of overlap between separate conversations), and the overall failure rate exhibited by the system. We hypothesize that the accuracy of the fault localization method will suffer under highly connected services, since the combined effect of the increases in the above factors should have a negative impact on the ranking position.

We present our results in Figure 7.18, where we compare the low and high connectivity scenarios using the same sort of plot as in Section 7.2.3. These results confirm our hypothesis: In the low-connectivity scenario, our method achieves significantly better accuracy for both ranking algorithms than in the high-connectivity scenario. The better result for low connectivity is due to the relatively small number of services involved in each conversation (on average, 2.03 versus 7.9), the relatively low degree of overlap between conversations, the relatively low dependence graph FP ratio (0.2% versus 3%). and a lower inherent failure rate (5.01% versus 20.88%). The reverse is true for the high-connectivity scenario.

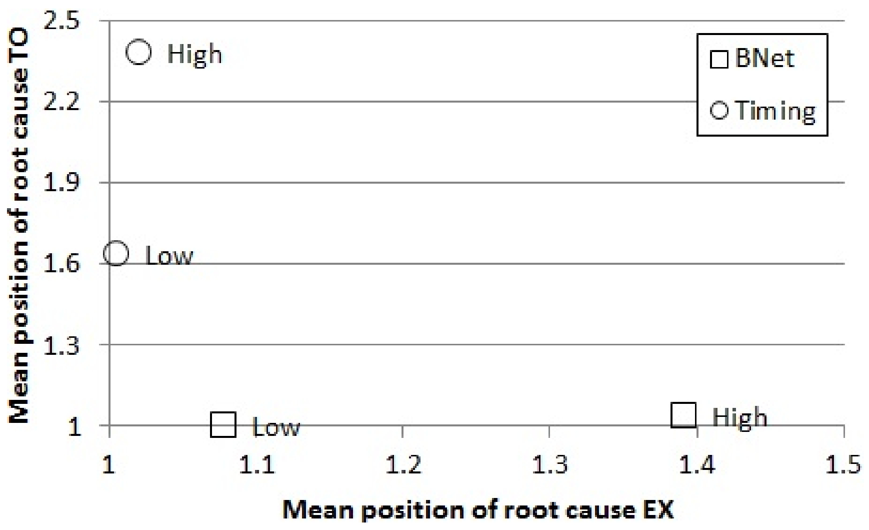


Figure 7.18.: Impact of service connectivity on mean ranking position of correct root-cause fault

Notice that in our other experiments we use the settings for a high service connectivity. This means that those results are based on a substantially more challenging scenario.

Note that we have also experimented with medium connectivity scenario having double the connectivity of low connectivity scenario and half connectivity of the high connectivity scenario (same as in the evaluation of the dependence discovery 7.1), however, the precision of the fault localization was almost the same as in the low connectivity scenario.

7.2.5. Impact of service fault rate

In our final experiment, we investigate the accuracy of our method under a range of service fault rates. In order to show impact of various fault rates we vary the rate at which services fail when invoked (i.e. receive request message). Unlike the failures caused by network faults, which reflect the underlying network conditions, the service fault rate allows us for a controlled experiment to see the performance of our fault localization method under wider range of the frequency that faults occur in the system. Recall that we configure the simulations such that when a client experiences a failure, 90% of the time the root cause is an exception and 10% of the time the root cause is a timeout. Note that it is important to maintain reasonable ratio of timeout faults in order to simulate the decreased reliability of mobile nodes. We expect that increasing fault rate should be negatively correlated with precision of localization of both types of faults.

The results are shown in Figures 7.19 and 7.20 for both ranking algorithms, under the high connectivity scenario, and a 6-second time window. The service fault rate is varied in the x-axis from 0% to 2%. For each rate, we also give the total number of failed conversations. Note that even with a 0% service fault rate, the scenario reflects failed conversations caused by network faults during the entire simulation.

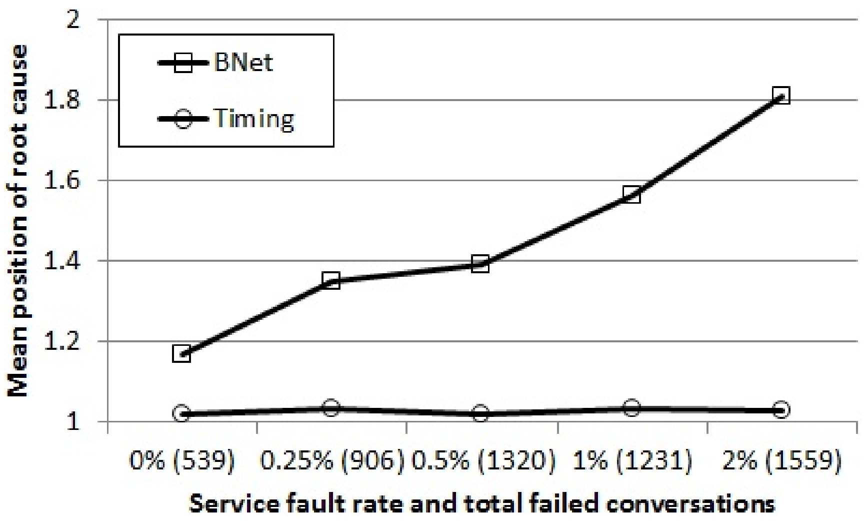


Figure 7.19.: Impact of service fault rate on mean position of correct root-cause exception faults. Both the fault rate and the total number of failed conversations for each rate are shown

The results demonstrate that the timing-based ranking algorithm is fairly robust in a wide range of service fault rates, maintaining consistently high accuracy even under high fault rates. The accuracy of BNet-based ranking, however, deteriorates as the service fault rate increases. This is because the BNet algorithm is sensitive to the size of the FPM, which increases not only due to inclusion of irrelevant dependencies (recall Figure 7.17), but also due to the increase in the fault rate, causing some irrelevant failures from overlapping conversations to be included in the FPM.

The Timing-based ranking deteriorates as well, however, in significantly lower rate since the effect of the much lower 10% proportion of timeout faults is minimal.

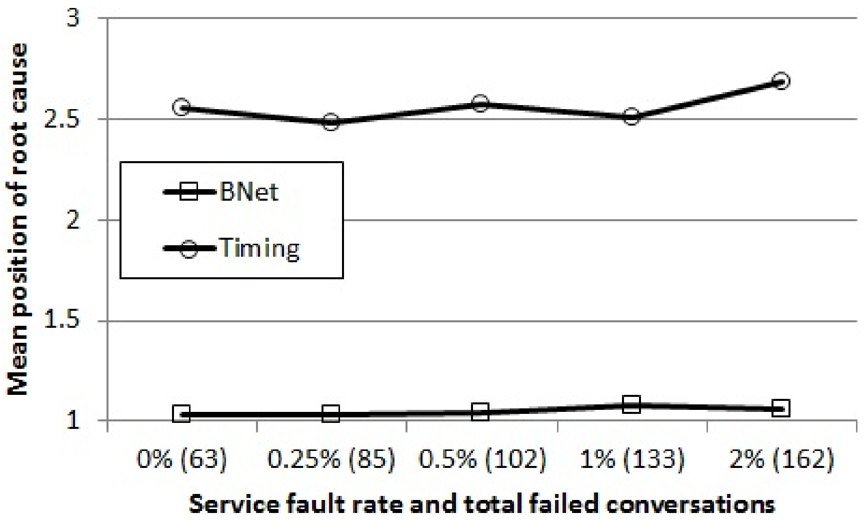


Figure 7.20.: Impact of service fault rate on mean position of correct root-cause timeout faults. Both the fault rate and the total number of failed conversations for each rate are shown

7.2.6. Estimates of data storage and data transfer needs

In this section we investigate the data storage requirements placed on the monitors, and the data transfer requirements placed on the network. (We do this by positing a representation for the data, analyzing the space requirements of that representation, and computing the total for a particular experimental scenario.)

A monitor does not require any storage space for data since the monitor is only inspecting system log for symptoms recorded by services and does not maintain any data on its own. The data transferred over the network to a fault localization element contains only aggregated data for the desired time window.

The data bundle transferred by an individual monitor contains data for one service and status of root cause modes relevant to the client's fault (i.e. SF and SENDF for EX fault and TO for TO fault). The status of each mode is aggregated into record containing Boolean value representing whether any relevant symptom occurred in the queried time window and timestamp of latest symptom (required for Time based ranking).

Consider the high connectivity scenario. The typical data bundle transferred by an individual monitor for EX fault contains data for one service and thus two modes. Thus the bundle contains two mode records, each consisting of Boolean value and timestamp. To transfer these data in SOAP message would require about 0.3 KB message. The high connectivity scenario has on average 7.9 services in DG thus average amount of data transferred over network would be about 2.5 KB per fault.

7.2.7. Discussion of results

The experimental results above demonstrate the effectiveness of our fault localization method in ranking the correct root causes of failed service conversations. In particular, the timing-based ranking is shown to be very effective in localizing the failures caused by service exceptions, while the Bayesian-based ranking is more effective in the case of timeout failures. This suggests these two algorithms can be separately employed according to the kind the failure reported by a client.

The results also show the importance of the proper selection of the time window size, where a value much smaller than the longest duration of the conversations is shown to provide good outcomes, reflecting a relatively quick propagation of the failures in the system, compared to the time it takes to complete the conversations. Moreover, the method is also shown to perform consistently in a wide range of rates at which service faults occur in the system.

An important consideration in the MANET environment is the requirement for data storage and transfer. We have shown that our method performs well in this regard.

The BNet inference performs quickly on laboratory PC–the construction of BNet from FPM and the inference of all root causes takes on average of about 3 seconds.

We conclude our evaluation with a look at how each aspect of our fault localization method contributes to the overall results. To do this, we break down the method into the contributions of its three main elements, each of which is applied in turn to obtain a result: (1) the dependence graph, (2) the fault propagation model, and (3) the ranking algorithm. To see the contribution of each element, we start from the total number of candidate root-cause faults observed within the selected time window. We then determine how many are eliminated at each step of the method. We use the high-connectivity scenario with 0.5% service fault rate and 60-second time window size. The larger window size allows us to see the reduction in each step more clearly; results for the 6-second window size show similar trends.

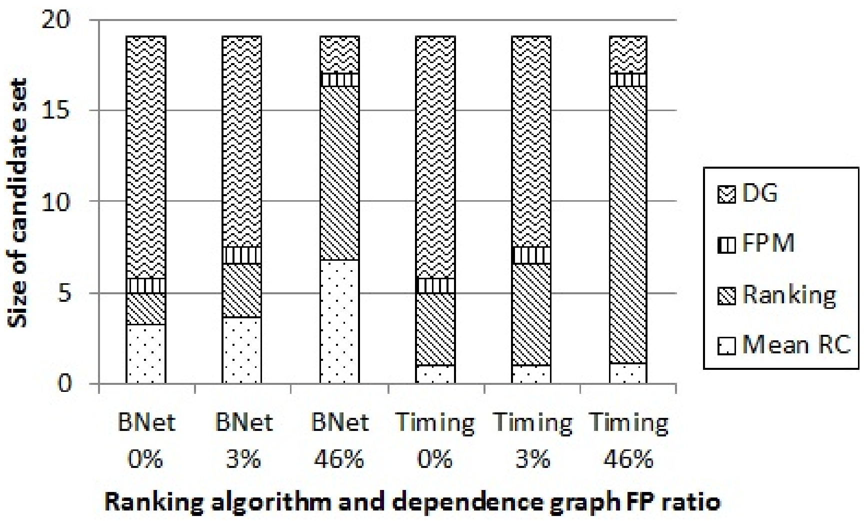


Figure 7.21.: Contribution of fault localization method steps to exception fault analysis

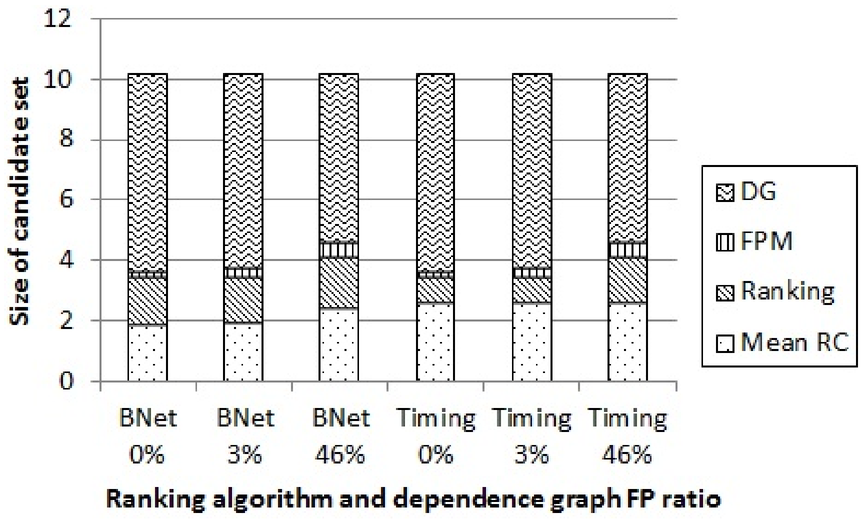


Figure 7.22.: Contribution of fault localization method steps to timeout fault analysis

The results are shown in Figures 7.21 and 7.22. The height of each bar represents the size of the candidate set. As noted above, there are 19 and 10 candidate root causes on average for the exception and timeout failures, respectively. (The total number of *symptoms* observed by the monitors is much higher—102 and 13 failure symptoms for exception and timeout failures, respectively—but as the monitors report only *aggregated* data, the number of candidates is much smaller.) Each section of the bar represents the number of candidates eliminated by an element of the method: the DG area represents the reduction to those services appearing in the dependence graph rooted at the client that witnessed the failure; the FPM area represents the reduction to those reachable to the client in the FPM; the Ranking area represents the reduction to those including the correct root cause; and the Mean RC area represents the final mean position of the correct root cause.

Overall, we can see that each element indeed makes a contribution. The specific contribution differs with the ranking algorithm and accuracy of the dependence graph. Indeed, the ranking algorithm and dependence graph play the major roles in localizing the root cause of client failures. The effect of DG accuracy is particularly obvious when the 46% FP ratio is compared against those of 0% and 3% for exception faults, suggesting that it is important to have an effective dependence discovery in diagnosing such failures. The timing-based ranking algorithm contributes a significant portion in narrowing down exception faults, even when the DG and FPM is not effective. The BNet-based ranking algorithm is more effective for timeout faults and less so for exception faults. Again, this supports our suggestion to apply the ranking algorithm most appropriate to the reported failure.

7.3. Evaluation of Dependence Discovery with Distributed Data Harvesting

We evaluate the dependence discovery method employed in conjunction with the distributed data harvesting method. Hence, the data used by the dependence discovery are harvested from backup stores maintained by the distributed data harvesting method instead from the direct on-demand access to monitors as in the experiments presented in Section 7.1.

The distributed data harvesting algorithm continuously synchronizes data between network nodes. The data stored in the backup stores by the distributed data harvesting algorithm are used by the discovery element in construction of the dependence graphs. The dependence graphs are constructed on demand by the discovery element. The graphs are rooted at a given client, beginning at a given time instant, and for some time window. Each dependence graph is constructed for a particular conversation; thus the time window begins and ends with start and end of that conversation. The data provided to the discovery element include both relevant and irrelevant information, since the local data backup store will provide data about *all* interactions traversing the involved services during the time window. These interactions involve not just those of the given conversation of interest, but also those of others.

Recall, that our method by design loses information and is sensitive to the dynamics of the operational environment. The monitors retain only aggregate data, not individual messages. Furthermore, the synchronization mechanism used to transfer the data between nodes, employs further data aggregation and thus causes additional loss of precision.

The evaluation questions of interest therefore center on the capability of the synchronization algorithm to transfer data from monitors to client nodes. In particular, we examine the factors that should influence capability of the method to transfer data and we look at few additional aspects of our method:

1. **Impact of synchronization frequency:** We examine the impact of the number of synchronization cycles on availability of the dependence data in the backup stores on the client nodes. We hypothesize that as the number of cycles between the end of theconversation and the start of the data harvesting is increasing, so too should the availability of the dependence data.

2. **Impact of number of peers:** We examine the impact of the number of peers selected during each synchronization cycle on availability of the dependence data in the backup stores on the client nodes. We hypothesize that increasing the number of peers per synchronization cycle should increase the availability of the dependence data.

3. **Optimal synchronization strategy:** The synchronization frequency and the number of peers per synchronization cycle are two parameters of the method which participate in synergy on the overall performance of the method. We will look for optimal combination of these two aspects of the method to provide the optimal strategy for use of our method.

4. **Comparison with other method:** We compare our method with the approach of direct access to monitors.

5. **Data storage and transfer requirements:** We examine the data storage and data transfer requirements of our method.

7.3.1. Evaluation metrics and parameters

Our experiments focus on a particular hypothetical conversation C. A good result for our method would be that it can transfer as many dependencies of C as possible, while not including the dependencies of other conversations. Same as in evaluation of the dependence discovery method in Section 7.1, we use two metrics to characterize the quality of our results, namely the ratio of true positives (TP) and the ratio of false positives (FP), as defined in Section 7.1.

A high TP ratio indicates that most of the dependencies in C have been transferred from the monitors to the client node. A high FP ratio indicates that the discovery result mistakenly includes a large number of dependencies of conversations other than C. These irrelevant dependencies need to be minimized in order to improve the accuracy of the DG.

In the evaluation of the dependence discovery method employed in conjunction with the distributed data harvesting method, we focus only on one type of the dependencies, namely inter-dependencies. It is because including both inter- and intra-dependencies does not add any additional insight into understanding of characteristics of the distributed data harvesting algorithm.

Our goal is to explore behavior of our method under various conditions in MANET networks. Thus, in the evaluation of the method we use two different scenarios. Both of these scenarios represent the same service system; however, each of them represents different type of mobile network.

The first scenario represents *military* unit on mission operating in open terrain. The unit consists of 50 members which are separated into several groups, each having about 10 members. The members of each group move together by walking speed across the terrain as they fulfill their mission. However, occasionally member leaves one group for another and thus travels between groups in open space. The groups operate in area with bounds of 2x2 kilometers. To simulate the unit members moves we use Nomadic Community Mobility Model. Given the limitations of the used Wifi standard and its signal range; devices of members of one group can communicate together for most of the time, however, the groups are disconnected from one another for almost all the time. Yet, as they pass though the space, they meet and can communicate together for limited periods of time. Each member has a mobile device with installed client application. Some of the devices host services which can be used by client applications or other services. Because the network is partitioned into several groups, each service has 5 identical instances hosted on 5 different nodes in order to provide maximum availability.

	Firefighting	Military
Spatial bounds	1km x 2km	2km x 2km
Mobility model	Random Waypoint	Nomadic Community

Table 7.6.: Scenario specific parameters

Number of nodes	50
Mobility speed	3 - 6.6 km/h
WiFi standard	80211b
WiFi unicast rate	11Mbps
WiFi multicast rate	1Mbps
Transmit power	-15 dBm
Path loss mode	2ray
Routing protocol	OLSR
Protocol stack	TCP/IPv4

Table 7.7.: Network-layer parameters

Number of clients	50 (one per node)
Number of services	25
Size of dependence graph	2 – 6
Invokable methods per service	2
Workload (client request rate)	30s
Number of service replicas	5
Response timeout	60s

Table 7.8.: Service-layer parameters

The second scenario represents *firefighting* unit fighting forest fire. The unit consists of 50 members representing either personnel or vehicles. The spatial bounds are set to 1x2 kilometers representing fire front line with area within which the unit operates. The unit members move across the area individually by walking speed with frequent stops as they carry on their firefighting activities. To simulate the moves of the unit members, we use Random Waypoint Mobility Model. Because the members move across the area in random fashion, the links between all of the nodes are in constant change. Therefore, this scenario represents challenging case with highly dynamic topology. Summary of the scenarios is provided in Table 7.6.

We use the service-based system emulator described in the Section 6.2 in the evaluation of the method. Table 7.7 summarizes the basic network-layer parameters used in our experiments. These are standard settings used widely in the networking community and embodied in the EMANE simulator. We use the IEEE 80211b Wifi standard, with 2ray path loss mode of wireless signal propagation. The transmit power is set to standard -15dBm and the transmit data rates are set to 11 Mbps for unicast and 1 Mbps for multicast. The different transmit rates of unicast and multicast, allow us to simulate the impact of routing protocol limitations. Another important parameter is the mobility speed of the mobile hosts. For all of the experiments we set the mobility speed of all nodes to rage of 3 to 6.6 km/h (0.833 to 1.833 m/s). This allow us to simulate behavior of network composed of nodes carried by walking people.

In Table 7.8 we provide the basic service-level parameters used in our simulations. The timeouts are set to default Glassfish/Metro value for SOA and Web Services implementations. Note that the number of methods in each service is not significant from a simulation point of view, as long as we have at least two methods available so that we can simulate the impact of overlapping conversations with different dependence graphs.

Same as in the simulator based experiments, for the interconnection model, we use a 2-tiered topology. The first tier consists of the connections between the clients and a set of "front end" services, while the second tier consists of the connections between the services themselves. In our experiments, we use 50 clients, five front-end services, and 20 "back end" services. When starting a conversation, each client invokes a method selected uniformly at random from all methods provided by the five front-end services.

In order to isolate the effect of the system complexity, we configure the service system such that in given scenario, each conversation involves specific number of services. In this way we control the size of the dependence graph. For most of the experiments we configure the scenarios to include four services in conversation. However, because certain proportion of conversations fails (i.e. due to limited network connectivity), the average number of services involved in conversations of particular scenario is smaller.

We collect our results from 40 minutes of execution run-time after excluding 10 minutes of warm up. Each combination of parameters in our experiments results in thousands of conversations occurring during the simulated 40-minute execution. The results given below are averages over the data collected from these conversations, where each conversation is then a statistical sample subject to the random variables.

7.3.2. Reachability of nodes in MANETs

In the experiments presented in the Sections 7.1 and 7.2, we have taken the assumption of complete availability of the dependence and symptom data on demand. However, in a real network environment the reachability of nodes within a mobile network is limited either due to the partitioning of the network or due to the limitations of the routing protocols.

To appreciate the seriousness of the reachability problem, consider Figure 7.23. The experiments examine two different mobility behaviors under the same hypothetical application. The firefighting behavior is characterized by independent movements of nodes, while the military is characterized by collective movements. A management node attempts to harvest the data describing the service interactions involved in a particular conversation. It does so by iteratively contacting the monitors associated with each node in the conversation, beginning with the root node (i.e., the client). In this way, it dynamically discovers the dependence graph. Depending on when the process is initiated, represented by the horizontal axis, a different percentage of MANET nodes, represented by the vertical axis, is in fact reachable. The "levels" correspond to the distance in hops from the root of the conversation. The experiments clearly illustrate the flaw in a direct harvesting method and motivate the design of the distributed data harvesting method.

When the DG is constructed, certain dependencies are harvested from local node of the client, while the rest is obtained from remote nodes. The locally harvested dependencies include the first dependence of the conversation between the client and the front-end service. Additionally, on nodes which host services, the local services may be involved in the clientŠs conversations and thus their dependencies are harvested locally as well. The dependence data of services hosted on the remote nodes are harvested on demand from the remote monitors. Given the reachability of the nodes shown in Figure 7.23, the discovered dependencies in the military scenario are shown in the Figure 7.24. The lower area represents the locally harvested dependencies. Out of four dependencies in the DG, about 33% are harvested locally. The rest of about 77% are harvested from remote monitors. The middle area represents dependencies successfully obtained from remote monitors with the on demand requests. The upper area represents portion of dependencies missing in the DG due to unreachability of the remote nodes. Thus, the discovered DG has about 10% to 25% of dependencies missing depending on a delay. In the firefighting scenario the reachability of nodes is lower and thus the discovered DG has 30% to 55% of dependencies missing in the DG depending on harvesting delay as shown in Figure 7.25.

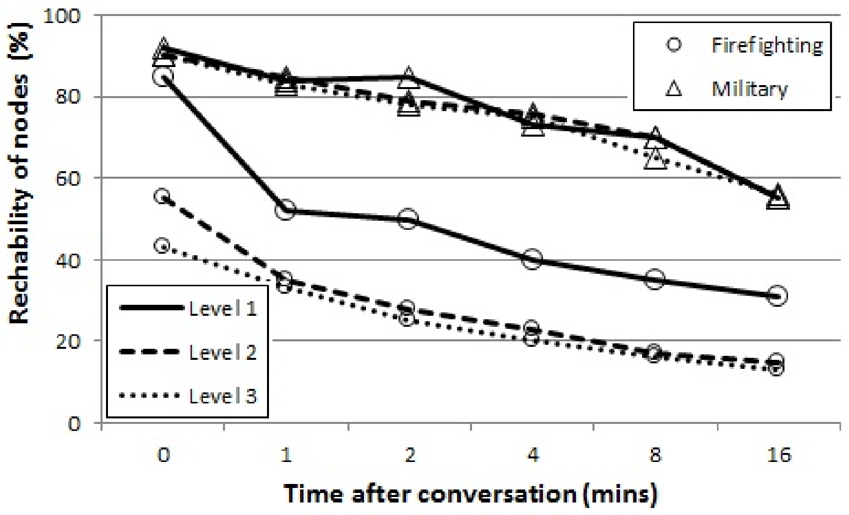


Figure 7.23.: Reachability of nodes in the firefighting and military scenarios

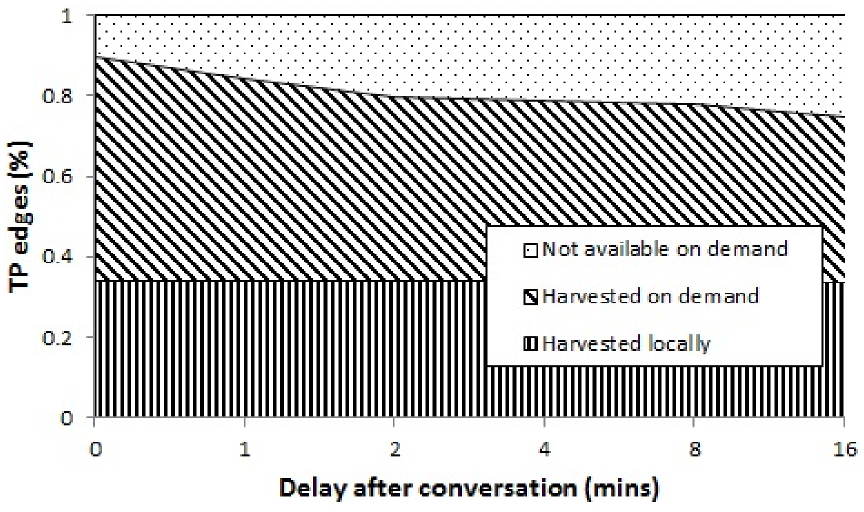


Figure 7.24.: Reachability of dependencies in the military scenario

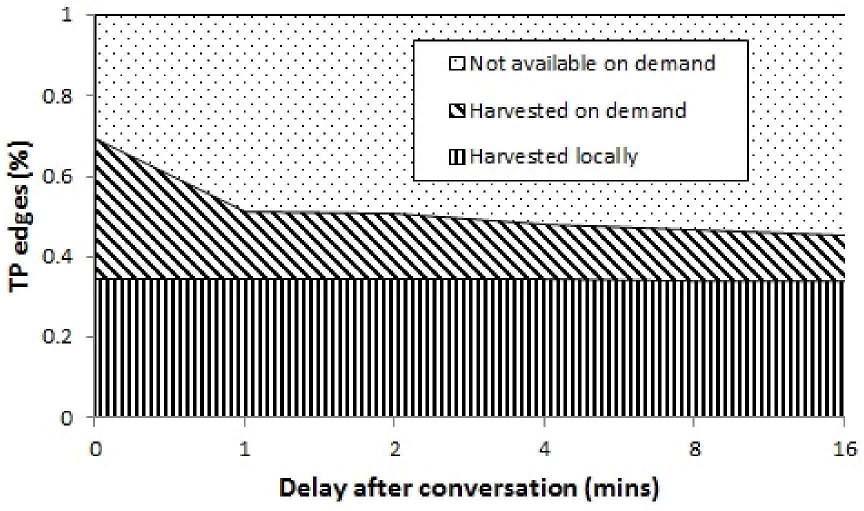


Figure 7.25.: Reachability of dependencies in the firefighting scenario

7.3.3. Impact of synchronization frequency

We first look at the propagation of the data from monitors to client nodes with single peer per synchronization cycle (i.e. pairwise gossip). This allows us to examine the basic characteristics of the algorithm. With single peer per synchronization cycle; in each cycle for each node is selected randomly single peer from one hop distant neighbors. Thus, ideally in each synchronization cycle, data propagate to one node one hop away from every node, which already hosts given data (i.e. the original source node of the data and every node, which already received the data).

We are interested in the impact of the increasing number of synchronization cycles on availability of the dependence data in backup stores on the client nodes. We hypothesize that as the number of cycles between the end of conversation and the start of the data harvesting is increasing, so too should the availability of the dependence data. The availability of the data can be measured as a ratio of TP dependencies in a DG. With no synchronization cycles occurring before the data harvesting, the only dependence data available in construction of the DG are harvested from the local monitors. However, with every next synchronization cycle the amount of data arriving from remote monitors should increase.

Note that the data do not need to achieve full network penetration in order to be available for construction of the DGs. In order to be available for harvesting from the backup stores, the data only need to be transferred from its source to the client nodes.

In this experiment we set the delay between the end of the conversation and the start of the dependence harvesting to 5 minutes. The results are reported in Figure 7.26. Initially with no synchronization cycles, the only data available are from the client monitors and from the locally hosted services. With increasing number of synchronization cycles, the backup stores are receiving increasing amount of dependence data from the remote monitors. In the military scenario, the availability of the data is increasing faster than in the firefighting scenario. It is because in the military scenario the services involved in the conversations are hosted on nodes within the same group as the client. Thus, the dependence data need to be transferred between fewer intermediate nodes then in the firefighting scenario. Whereas in the firefighting scenario, the services involved in the conversations can be drawn from higher distances as the nodes are dispersed relatively evenly over the large area. Therefore, to transfer the data from more distant nodes, higher number of synchronization cycles is required.

In both of the scenarios after 8 synchronization cycles the availability reaches about 90%. With further cycles, the availability is increasing only slowly as data arrive from nodes further afield. Eventually, with 32 cycles the availability reaches 97% in the military scenario and over 99% in the firefighting scenario. The lower availability in the military scenario is caused by the migration of the nodes; as some nodes disconnect from one group for another, the connection is lost for a long period of time and thus small fraction of data does not arrive regardless of the number of cycles. This is not the case in the firefighting scenario and thus the availability of the data is steadily increasing and eventually reaches almost 100%.

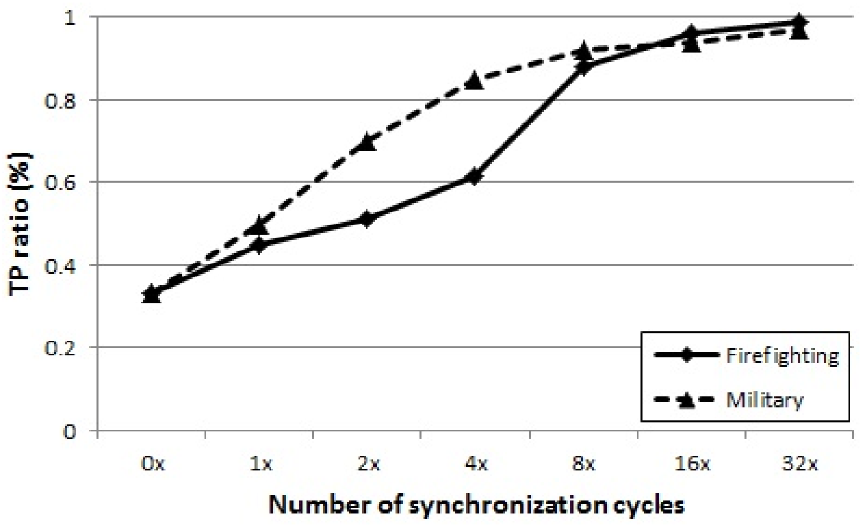


Figure 7.26.: Availability of dependence data with different number of synchronization cycles given as TP ratios

7.3.4. Impact of number of peers

We now investigate the impact of multiple peers used in the synchronization. In this experiment, we vary the upper limit of number of peers selected during each synchronization cycle. Same as in the previous experiment, the data harvesting is started five minutes after a conversation ended. However, in order to isolate impact of the variable number of peers, we fix the synchronization frequency to one per minute. Thus, there are at least four synchronization cycles initiated by each node after the conversation concluded and before the data harvesting starts.

The number of selected peers is limited by two factors. First, only one hop distant neighbors are considered as peer candidates. Second, from the peer candidate options only certain number limited by the configured upper limit is randomly selected. Consequently, the number of selected peers cannot be higher than either of these two limits, and therefore, the average number of the selected peers will be somewhat lower than the configured limit.

In this experiment we are interested to see the impact of increasing of the upper limit of the number of peers on availability of data in the backup stores. We hypothesize that increasing the number of peers should increase availability of the dependence data. The results are reported in Figure 7.27. As expected, with increasing number of peers per synchronization cycle, the availability of the data is increasing. However, upper limit of about 5 peers seems to be the threshold above which further increases in number of the peers do not have any additional positive impact. The threshold limit is an important information in configuration of the method, because it allows us to control the network overhead imposed by the method.

Furthermore, notice that in comparison with the single peer approach per synchronization cycle presented in Figure 7.26 in the previous section, the multiple peers per cycle provide somewhat better outcome. For example five peers per cycle provide 99.4% and 98.6% of data in military and in firefighting scenarios respectively. However, the single peer provides only 94% and 96.3% of data with same number of synchronization cycles in total. It is because all peers are selected randomly at every synchronization cycle. Thus when single peer is selected, there is same probability to select peer which was either selected in previous cycle or from which data were received recently. Thus the effectivity of the data dispersion is decreased. This is particularly pronounced when node has few neighbors. However, with multiple peers per cycle this problem is reduced since several different peers are synchronized at one time. Therefore, the multiple peers per cycle approach provides better synchronization strategy than single peer per cycle.

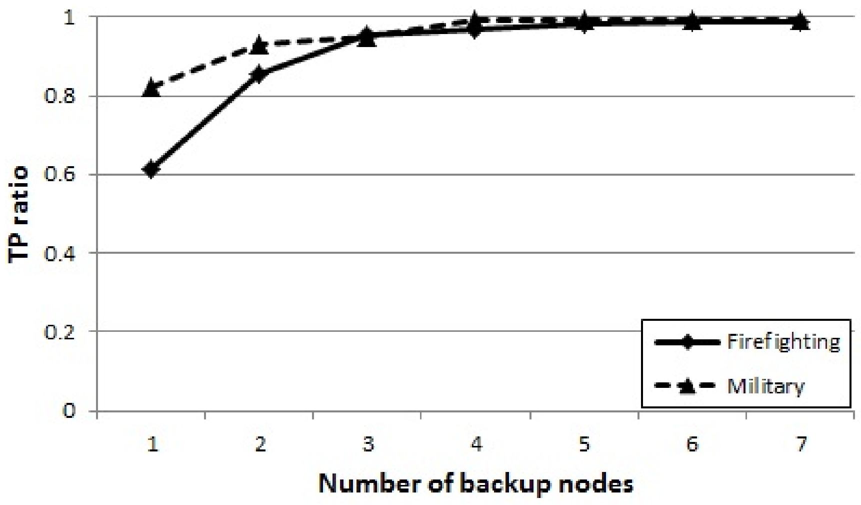


Figure 7.27.: Availability of dependence data with different number of peers per synchronization cycle given as TP ratios

7.3.5. Comparison of harvesting methods

We now compare the performance of the distributed data harvesting approach with the direct on-demand data harvesting approach.

The TP ratio achieved with the direct method is presented in Section 7.3.2 and shown in Figures 7.24 and 7.25. For example, with 5 minutes delay of the data harvesting, the direct method yields about 79% and 48% TP ratio in the Military and the Firefighting scenarios respectively. The distributed method with five peers per synchronization cycle and one cycle per minute yields 99.4% and 98.2% TP ratio as shown in Figure 7.27.

In comparison of the FP ratio, we modify the experiments in order to provide a meaningful comparison scenarios. We configure the mobile network to have a full connectivity between all nodes for evaluation of the direct method. In the fully connected network, the direct method can discover whole DGs unlike in the partially connected mobile network. In this way we can compare the FP ratios of DGs produced by the direct method with FP ratios produced by the distributed method in the partially connected mobile network.

The FP ratio of the direct method is product of two factors. First factor is the aggregation of the data by the monitors which loses some detail due to aggregating the dependence occurrences into the time slots. In this experiment the monitors use size of time slot of 0.1 second, which provides good tradeoff between data resolution against memory consumption. Second factor is the size of the time window from which dependencies in the DG are included. The size in all experiments is based on the duration of the conversation; thus allowing to include all TP dependencies and limit inclusion of FP dependencies to a minimum. The distributed method uses an additional aggregation of the dependence data in order to transfer the data efficiently over the network. Thus, we hypothesize that the additional aggregation of the data into coarse grained time slots causes further loss of precision and further increases the FP ratio.

Figure 7.28 presents our results. The Direct method is represented by a single value of 16.2% of the FP ratio. We provide only a single value because in the fully connected network experiment, mobility of the nodes has not impact on behavior of the service system. For the distributed method, we provide results for three different time slot sizes used in the synchronization data transfer. The slot size of 0.1 second yields 16.4% and 18.2% FP ratios. These results are comparable with the direct method in both of the scenarios. The somewhat higher FP ratios in both of the mobile networks are caused by conversations having smaller DG size, because not all services configured to be invoked in the conversations are reachable due to network failures. Thus the DGs contain higher proportion of intermediate services with higher conversations overlap and thus higher FP ratios. As the size of the time slot used in the data transfer increases, so too is increases the FP ratio. Thus, with a size of time slot of 10 seconds the FP ratio reaches 35% and 37% in the military and the firefighting scenarios respectively. The difference in these ratios is caused by the different network topology impacting service selection and conversation overlap.

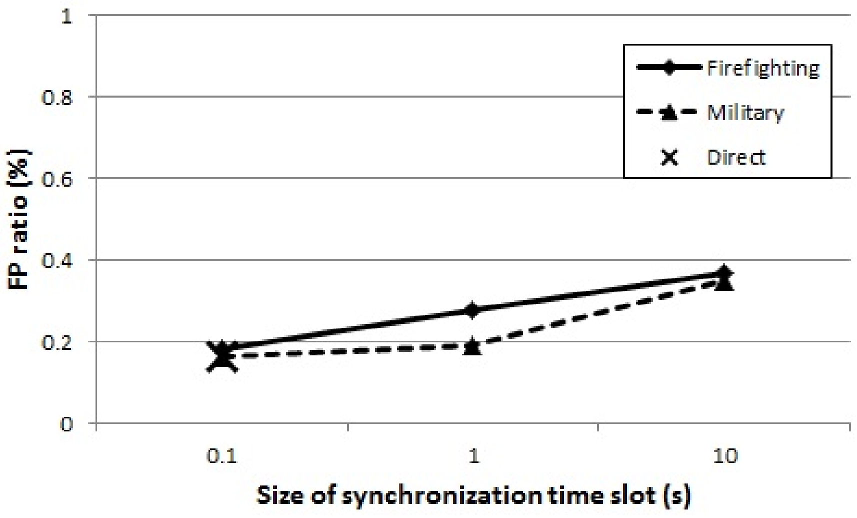


Figure 7.28.: Precision of dependence graph with different size of synchronization time slot given as FP ratio

7.3.6. Network overhead and data storage needs

The continuous data synchronization process imposes constant overhead on the underlying physical network. In Sections 7.3.3 and 7.3.4, we have established the relationship between the synchronization frequency and the number of peers per cycle. Figure 7.29 illustrates the network overhead as an amount of data transferred with variable number of peers per synchronization cycle with fixed frequency in the firefighting scenario.

In synchronization of peer, the data sent to the peer contain all changes in the local data as well as all the new data received since the last successful synchronization with the peer. Thus, with increasing number of peers per cycle, the amount of data sent to each individual peer is increasing, since more data were received and accumulated from other nodes between the synchronization cycles. However, with more than eight peers per cycle, the amount of data sent to each peer is somewhat decreasing since almost all of the peers are synchronized in each cycle and thus less changes accumulate between cycles.

We can compare the alternative synchronization strategies (i.e. combinations of synchronization frequency and number of peers per cycle) based on the total number of synchronizations to peers in fixed period of time the strategy requires to achieve certain comparable data availability in the backup stores. In the scenario with single peer per cycle, with 32 synchronization cycles the data availability reaches 97% and 99% of data in the military and firefighting scenarios respectively. While with five peers per cycle and four synchronization cycles the data availability reaches 99.4% and 98.6% of data in the military and firefighting scenarios respectively. Thus, given the size of the data transferred per cycle shown in Figure 7.29, the single peer based scenario will transfer total of about 10 kB of data per node in all of the 32 cycles. While in the five peers based scenario the total amount of data transferred will be about 22 kB per node in all of the four cycles. Thus we can conclude that synchronization strategy based on lower number of peers per cycle, provides better performance profile in terms of network data overhead than the multiple backup strategy.

Another important aspect of the method is the storage needs of the data in backup stores on nodes. The data stored in backup stores are regularly pruned to limit the amount of data stored to minimum. The data stored are limited to certain maximum age and obsolete data are removed. The total amount of data stored depends on complexity of the system. For example, in the military and firefighting scenarios the amount of data stored on individual nodes is at most 25kB, with data older than 20 minutes being pruned.

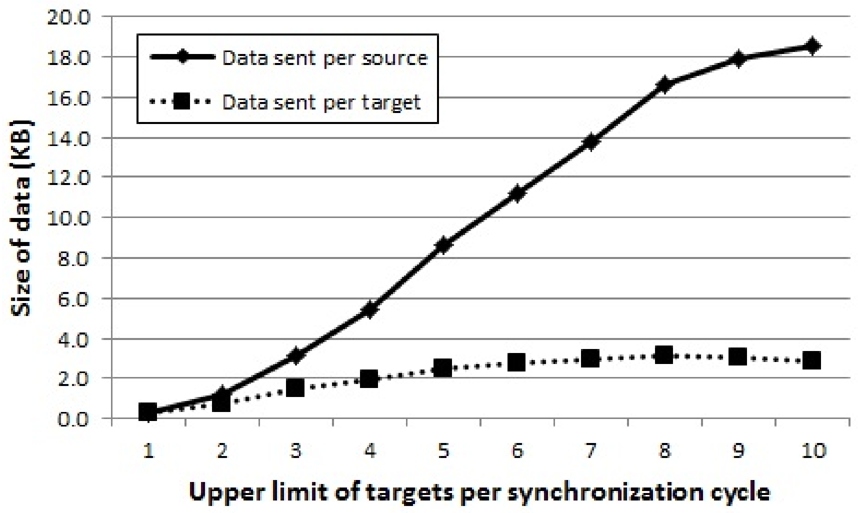


Figure 7.29.: The size of data transfered between source and target nodes with different number of synchronization peers per cycle

7.3.7. Discussion of results

The experiments presented above establish how the synchronization frequency and the number of peers per synchronization cycle affect the capacity of the distributed data harvesting method to transfer dependence data from monitors to client nodes. The capacity of the method to transfer the dependence data is also affected by the characteristics of the network; such as the degree of partitioning and the spatial-temporal behavior of the mobile nodes as is evident from the differences of data availability achieved in the various scenarios. This represents a broad exploration of the behavioral characteristics of our method.

In general, the results indicate consistently the capability of the method to transfer data between nodes which are not directly connected as measured in terms of TP ratio. The method provides alternative synchronization options in selecting combination of synchronization frequency and number of peers per synchronization cycle. Both strategies of either selecting a high frequency with low number of peers or a low frequency with higher number of targets provide similar outcome measured in terms of TP ratios.

The comparison with the direct on demand access to monitors shows that the distributed approach can provide similar accuracy (measured in terms of FP ratio) of the data yet with significantly higher data availability. The aggregation of the data during the synchronization may increase the FP ratio, however, we have shown that with selection of a small size of the synchronization time slot, the impact is minimal.

The network overhead caused by the continuous data synchronization is an important factor in operation of the method. We have demonstrated that by selecting suitable synchronization strategy the network overhead can be minimized. The synchronization strategy based on the high synchronization frequency with lower number of peers causes lower network overhead than the low synchronization frequency with higher number of peers yielding comparable availability of data.

An important aspect of the method is the determination of parameters for optimal availability of data in backup stores in particular type of network and service system. Regardless of the synchronization strategy employed, the optimal threshold can be determined by measuring the average size of a DG while altering the synchronization parameters. The optimal threshold of the method configuration is achieved when the DG size stabilizes and does not increases with increases in either the synchronization frequency or the number of peers per cycle.

7.4. Evaluation of Fault Localization with Distributed Data Harvesting

We evaluate the fault localization method employed in conjunction with the distributed data harvesting method used instead of the direct access to the monitors. Same as in the evaluation of the fault localization with the direct on-demand data harvesting approach presented in Section 7.2, the evaluation questions of interest center mainly on the accuracy of the ranking of root-cause candidates. A good result, of course, would be that our method consistently ranks the correct root-cause candidate at or near the top. We examine the ranking accuracy in context of an impact of the precision of the dependence and symptom data the method receives from the distributed data harvesting method. In particular, we examine the following factors that might influence accuracy:

1. **Ranking algorithm:** We examine accuracy of the two ranking algorithms, timing and BNet, in terms of the position at which they place the correct root-cause failure.
2. **Impact of size of synchronization time slot:** We examine the sensitivity of the ranking algorithms to the size of the time slot used by the distributed data harvesting method to transfer data between nodes. We would expect that as the size of the time

slot increases, the transferred data become less precise, and the ranking algorithms might have difficulty maintaining consistent results.

3. **Data storage and transfer requirements:** We examine the data storage requirements and the overhead the data synchronization imposes on the physical network.

In the evaluation of the method we use identical scenarios as in evaluation of the dependence discovery method with the distributed data harvesting method presented in Section 7.3. Thus, we use both the military and the firefighting scenarios and with the same service and network layer settings. The service interconnections include four services in all conversations and the results presented are based on the DGs constructed from interdependencies only. The methodology used in the evaluation of the ranking algorithms is same as the evaluation of fault Localization presented in Section 7.2.1.

There are several differences between scenarios produced by the NS-3 based simulator and the CORE/EMANE based emulator. Most notable is the difference in the ratio of fault types occurring in the scenarios. While in the NS-3 based experiments the ratio of exceptions to timeouts received by clients is about 9 to 1 on average. The ratio in the CORE/EMANE based experiments is about 3 to 7 on average. This is due to multitude of differences in all of the experimental components. Perhaps most important is the difference in the simulation algorithms used in the MAC and PHY network layers of the simulators. These network layers are more realistically implemented in the EMANE and thus results from this experimental suit should be considered of a higher trustworthiness.

7.4.1. Impact of size of synchronization time slot

We first compare the accuracy of the two ranking algorithms used in conjunction with the distributed data harvesting method. In Section 7.2 we have established the accuracy of the algorithms for the two types of the faults found in the service-based systems. In this experiment, we investigate the sensitivity of the algorithms to the precision of the data available from the distributed method. The precision of the data available is affected by the size of the time slot used in the data synchronization. We hypothesize that the increasing size of the time slot used in the data synchronization will have a negative correlation with the accuracy of the ranking algorithms.

The results are shown in Figures 7.30 and 7.31. Only results for the firefighting scenario are provided because the results for military scenario are almost identical. We compare precision of the ranking algorithms and show the precision relatively to the number of options presented to the ranking algorithms and to the random guess. Same as in the NS-3 based experiments, the Timing algorithm is more effective in raking the EX faults and the BNet algorithm is more effective in ranking the TO faults.

These results confirm our hypothesis, the size of the time slot has a negative correlation with the accuracy of both of the ranking algorithms. The increasing size of the time slots has several negative impacts on the fault localization. First, the input into the fault localization is a DG. The FP ratio of the DG is negatively impacted by the increasing size of the time slot as presented in Section 7.3. The increasing FP ratio increases the size of the DG, which in turn increases inclusion of an irrelevant symptoms in a FPM. Furthermore, the increasing size of the time slot decreases a resolution of the symptoms data and thus further increases the inclusion of the irrelevant symptoms in the FPM. Higher amount of symptoms in the FPM increases the number of options presented to the ranking and thus decreases the precision of the ranking algorithms. The impact is more pronounced in the ranking of the EX faults than of the TO faults. It is because the time window used in construction of the FPM of TO faults is significantly larger than that for the EX faults. Thus the FPM already contains higher ratio of the FP dependencies and symptoms and is thus less sensitive to the inclusion of the additional FP data.

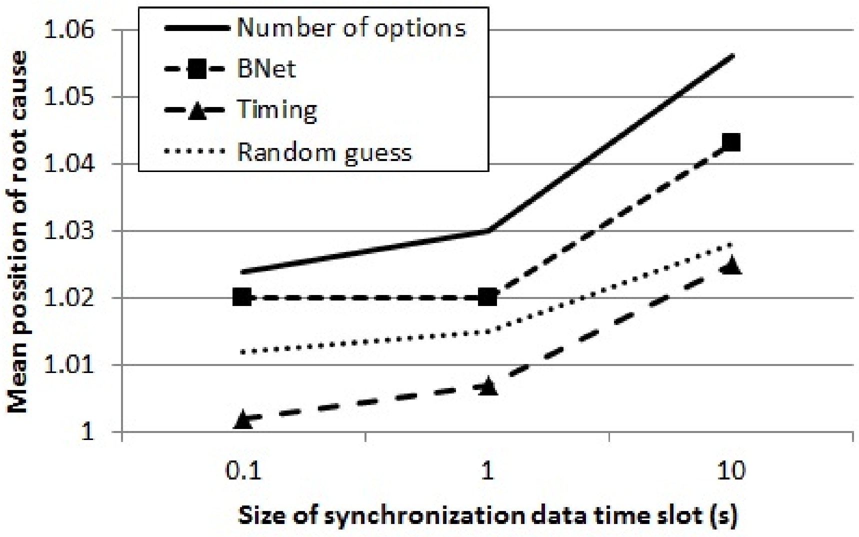


Figure 7.30.: Accuracy of ranking algorithms in ranking exceptions with variable time slot size

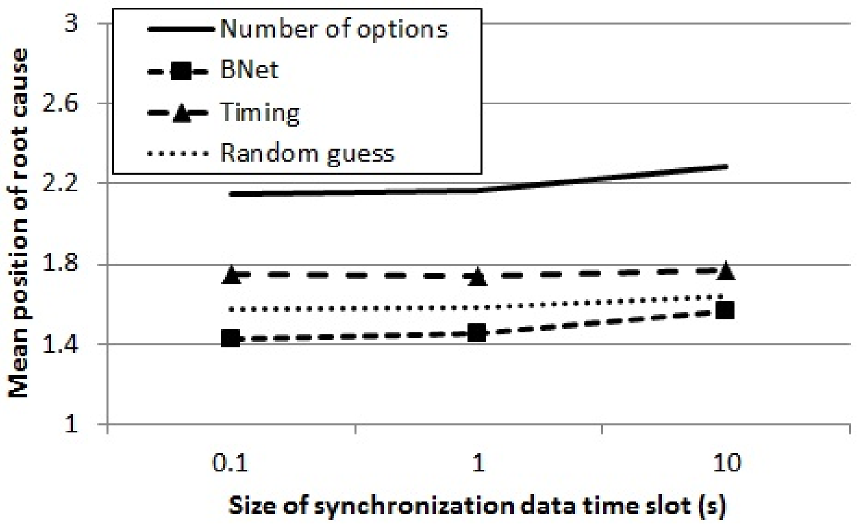


Figure 7.31.: Accuracy of ranking algorithms in ranking timeouts with variable time slot size

7.4.2. Network overhead and data storage requirements

In this section we investigate the data storage requirements placed on the backup data stores, and the data transfer overhead requirements placed on the network.

In Section 7.3.6 we have presented the overhead the continuous data synchronization places on the physical network while synchronizing the dependence data. The symptom data required for the Fault Localization method are added into the synchronization process as an additional source of the time series for synchronization. The symptom data are aggregated into the time slots same as the dependence data and with the same precision (i.e. the time slot size). Thus each of the additional time series should increase proportionally the overall amount of the data transferred.

The amount of data transferred depends on complexity of the system, size of the network as well as on the reachability of nodes within the network. The amount of data is also related to the size of the time slot used in the data synchronization. In Figure 7.32 is shown the sensitivity of the data overhead to the size of the synchronization time slot in both of the military and the firefighting scenarios with single peer per synchronization cycle. The overall amount of th data synchronized is increased by inclusion of the symptom data of about 20%. The size of the time slot has negative correlation with the overall amount of the data transferred i.e. as the time slot is increasing the overall amount of the transferred data is decreasing. However, as presented in previous section in Figures 7.30 and 7.31, the increasing size of the time slot has also negative correlation with the precision of the ranking algorithms. The increase in the amount of the transferred data with the increasing precision of the data (i.e. the decrease size of time slot) is however disproportionately slower. Thus for example in the firefighting scenario the 100 fold increase in the data precision from 10s to 0.1s of size of the time slot, increases the amount of the data transferred only by about 20% from 0.35kB to 0.43kB on average per node and cycle.

Correspondingly, with increase in the data transferred over the network is also increasing the size of the data stored in the backup stores. Thus in both of the military and the firefighting scenarios, the size of the individual store on each of the nodes increases from maximum of 25kB of the dependence data alone to maximum of 30kB for the dependence and the symptom data combined.

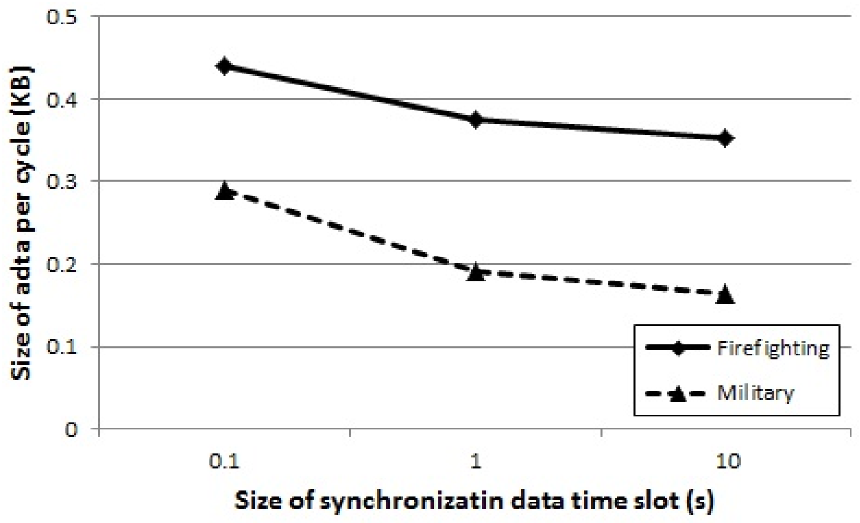


Figure 7.32.: Impact of size of synchronization time slot on amount of data transferred in synchronization cycle between pair of nodes

7.4.3. Discussion of results

The experimental results above demonstrate the effectiveness of our distributed data harvesting method in transferring symptom and dependence data for use in the fault localization. We have presented the accuracy of the ranking algorithms with various precision of the data available from the data synchronization. The accuracy of the ranking algorithms seems to respond well to the decreasing precision of data aggregated by the data synchronization.

An important consideration in the MANET environment is the requirement for data storage and transfer. We have shown that our method performs well in this regard. The network overhead is rising only very slowly with increase in precision of the data transferred. Likewise, the symptom data cause only small increase in the overall network overhead imposed by the data synchronization.

Finally, in these experiments we have demonstrated the effectiveness of the fault localization method implemented in the realistic environment of the CORE and EMANE, with the fault localization components implemented in the Java EE.

7.5. Threats to Experimental Validity

The results reported are a selection from the experiments we have conducted. This selection focuses on particular aspects of the method, as detailed above. The results withheld support the results reported.

The threats to the validity of the results derive from the prerequisites listed in Section 1 and the experimental parameter values detailed in each of the evaluation sections. We have chosen these values based on the joint experiences of the NS-3, CORE, EMANE and Java EE community of users, and on the recommendations of commercial SOA and Web Services providers. Where specific community experience was lacking in the choice of values (e.g., the absence of a benchmark) we attempted to broadly sample in the space of values. We made use of uniformly random distributions in several instances (e.g., the request behavior of clients), which are a simplification that is a methodologically accepted practice at this stage of investigation. This yields statistically sound results, as each data point in our results is an average over all arising conversations, each of which is a statistical sample subject to the random variables.

We make no claim that the results generalize beyond the reported experiments, but overall they give us confidence that the methods are a viable approach to service dependence discovery and fault localization in the challenging MANET environment.

8. Conclusion

We have presented suite of three run-time methods to discover dependencies, perform fault localization and to harvest data in service-based systems hosted on mobile ad hoc networks. The methods are designed to operate in the highly dynamic and resource-constrained environment of the MANETs.

The dependence discovery method discovers the dependencies among services operated in the highly dynamic environment of MANETs. Unlike existing approaches, the method does not require stable dependence relationships, nor does it require that large amounts of evidence data be collected over long periods. Through an extensive set of simulation-based experiments, we have evaluated the accuracy of the method in terms of operational factors characteristic of both service-based systems and MANETs. The method exhibits good behavior when subjected to the stress of a changing underlying network topology. Furthermore, its data storage and data transfer requirements scale well with the number and connectivity of the services involved.

The fault localization method analyzes the failures experienced and reported by clients of the service-based system in order to locate the root-cause fault, making use of symptoms observed at both the service and network levels. To cope with the temporal aspects of the problem, the method uses multiple elements to filter out irrelevant symptoms. These include a dynamic dependence graph rooted at the client produced by the dependence discovery method, a fault propagation model, and algorithms to rank candidate root causes. An extensive set of simulation-based experiments demonstrates that our method achieves fault localization results with high accuracy in a range of situations. It does so while incurring low data storage and transfer costs, making it ideally suited to the resource-constrained MANET environment.

The distributed data harvesting method uses an epidemic protocol to create a network wide synchronization data overlay to transfer the dependence and the symptom data over the unreliable links of teh MANETs. To overcome the limited connectivity of the MANETŠs nodes, the epidemic protocol transfers the data over intermediate nodes in successive synchronization cycles. The method minimizes the overhead caused by the continuous data transfer by imposing limits on the age of the data transferred and by eliminating irrelevant data. Through extensive set of emulation-based experiments, we have evaluated the capacity of the method to transfer th data from monitors to client nodes in various types of the MANET environments. We demonstrated that the method has small impact on the accuracy of the produced dependence graphs and on the fault localization while providing high availability of the data. Moreover, we have shown how to tune the method to optimize the overhead to minimum in the resource-constrained MANET environment.

Aside of the three methods, we have presented suite of experimental tools for analysis of the service-based systems hosted in MANETs. We have designed a new simulator of service-based systems hosted in MANETs. The simulator is built as an extension of a standard packet based network simulator NS-3. The simulator closely replicates the complex network behavior as well as the service-based system entities and models. We have used the simulator for evaluation of the dependence discovery and the fault localization methods. Furthermore, we have designed a generic web service system implemented in a Java EE. The system allows analysis of various types and configurations of web service systems in emulation-based or in real world environments. We have used the generic web service system in emulation-based evaluation of all of the three methods.

8.1. Applicability of the Method

Each of the methods makes certain assumptions about the environment within which operates.

The applicability of the dependence discovery method in a given system depends on information available in messages exchanged between services of the system. In systems which use plain message exchange protocols such as SOAP or REST, the available information is limited to identifier of the target service in request messages and therefore the discoverable dependencies are limited to the outgoing inter-dependencies. This approach is applicable in all service-based systems. However, in system where additional fields with source service and conversation-identifier are present in the messages, it is possible to discover intra-dependencies as well. This approach is applicable only in systems, which provide the necessary information in request messages such as those which implement SOAP extension standards WS-Addressing, WS-SecureConversation and WS-Coordination or similar extensions of other protocols. It is important to note that both of the approaches do not require any alterations of the system components or infrastructure except deployment of the monitors.

The fault localization method itself requires system components and services to report symptoms of faults into the system log. Furthermore, the symptom monitor requires permission to access the log to analyze the records. However, since in operation environment this approach might be unreliable source of symptom data (i.e. the configuration of the system and services might be difficult or some symptoms might not be reported at all). Thus, in Section 8.2 we provide overview of our work in progress, providing more realistic fault detection approach providing a more accurate symptom data.

To employ the epidemic protocol for harvesting of the dependence and symptom data, each node within the network should host an instance of the synchronization agent. It is important to note that none of the methods require any alterations of the system components or infrastructure except deployment of the monitors and agents on nodes.

8.2. Current and Future Work

Currently, we are working on a new fault detection method, which will allow detection of symptoms of faults from message flows in service-based systems.

In our current architecture, the system components, application environment and services have to be reporting all of the relevant fault symptom data into the logs. However, in the various operational environments, the capability to record the symptom data might not be available in all of the components and services. Moreover, the records stored into the logs have to contain all of the required information fields in order for the symptom monitors to access and correctly interpret the required information. Since logging is generally disparate and inaccurate across various system layers and types of components, some more reliable and accurate mechanism of fault detection is required.

In the new fault detection method, we focus on detection of symptoms of faults from flows of message, occurring at various locations in the service-based system. The method uses set of *passive detectors*, each responsible for detection of faults of a particular type of component of the service-based system. For example, detection of network faults by observing flows and exceptions issuing in application platform, or detection of timeouts by measuring response times.

This approach allows to collect all the symptom types available from the system logs as well as additional types of faults and observations not available otherwise. For example, this approach will allow us to probabilistically differentiate between root cause and transitive timeout symptoms. (The inability to differentiate between these two is currently source of lower precision of the analysis of the timeouts.) Hence, we expect, that the availability of the additional types of faults and data will allow us to further improve the fault localization method.

9. Appendices

A. Algorithms

A.1. Dependence Discovery: Construction of Dependence Graph

```
public DependenceGraph constructDG(
        String rootNodeURI,
        long fromTimestamp,
        long toTimestamp)
{
    DependenceGraph      dg = new DependenceGraph(rootNodeURI);
    int                  nodeIndex = 0;
    Node                 node;
    String[]             nodeDependencies;

    while ((node = dg.getNodes()[nodeIndex]) != null)
    {
        nodeDependencies = getInterDependenciesForNode(
                node, fromTimestamp, toTimestamp);

        //creates edges and target nodes in DG
        //if they do not exist yet
        addNodeEdges(node, nodeDependencies);
        nodeIndex++;
    }
    return dg;
}
```

A.2. Fault Localization: Construction of Fault Propagation Model

```
// creates FPM from DG and FPP
public FaultPropagationModel createFPM(
        FaultPropagationPattern fpp,
        DependenceGraph dg)
{
    FaultPropagationModel       fpm = new FaultPropagationModel();


    //create root node in FPM
    fpm.createRoot(dg.getRoot(), fpp.getRootMode());

    // add recursively all DG nodes into the FPM
    addSubNodesToFPM(fpm, fpp, dg.getRoot());

    return fpm;
}

// recursive - add all DG childern nodes of a DG parent node into FPM
private void addSubNodesToFPM(
        FaultPropagationModel fpm,
        FaultPropagationPattern fpp,
        DGNode dg_parent)
{
    for(DGNode dg_child : dg_parent.getSubNodes())
    {
        // add DG node into FPM
        addDGNodeIntoFPM(fpm, fpp, dg_parent, dg_child);

        // if DG node is intermediate - add its childern
        if (dg_child.isIntermediate())
        {
            addSubNodesToFPM(fpm, fpp, dg_child);
        }
    }
}
```

```
// add DG node into FPM and create propagation paths
private void addDGNodeIntoFPM(
        FaultPropagationModel fpm,
        FaultPropagationPattern fpp,
        DGNode dg_parent,
        DGNode dg_child)
{
    FaultPropagationPatternMode     fpm_parent;
    FaultPropagationPatternMode     fpm_child;


    // for each mode in FPP create new node in FPM
    for (FaultPropagationPatternMode fpp_child : fpp)
    {
        // for DG leaf nodes - add only root cause modes
        if (dg_child.isLeaf() && fpp_child.isTransitive())
        {
            continue;
        }

        fpm_child = fpm.createNode(dg_child, fpp_child);

        // link the new FPM node with its FPM parent nodes
        // add link to each transitive mode of parent
        for (FaultPropagationPatternMode fpp_parent : fpp)
        {
            if (fpp_parent.isTransitive())
            {
                fpm_parent = fpm.getNode(dg_parent, fpp_parent);
                fpm.createPropagationPath(fpm_child, fpm_parent);
            }
        }
    }
}
```

A.3. Fault Localization: Reduction of Fault Propagation Model

```
// reduces FPM based on symptoms
public FaultPropagationModel reduceFPM(
        FaultPropagationModel fpm,
        SymptomProvider sp)
{
    // assign symptoms to fpm
    assignSymptoms(fpm, sp);

    // remove nodes without symptoms
    removeNodesWithoutSymptom(fpm);

    // remove irrelevant nodes
    removeIrrelevantNodes(fpm);

    return fpm;
}

// harvest and assign symptoms to nodes of FPM
private void assignSymptoms(
        FaultPropagationModel fpm,
        SymptomProvider sp)
{
    long symptomTimestamp;


    for(FaultPropagationModelNode node : fpm.getNodes())
    {
        // harvest timestamp of symptom (if there was one)
        symptomTimestamp = sp.getSymptomTimestamp(
                node.getService(),
                node.getSymptomType());

        // assign the symptom timestamp
        if (symptomTimestamp != 0)
        {
            node.setSymptomTimestamp(symptomTimestamp);
        }
    }
}

// remove all nodes which have no assigned symptom
private void removeNodesWithoutSymptom(FaultPropagationModel fpm)
{
    for(FaultPropagationModelNode node : fpm.getNodes())
    {
        if (!node.getHasSymptom())
        {
            fpm.removeNode(node);
        }
    }
}

private void removeIrrelevantNodes(FaultPropagationModel fpm)
{
    boolean anyChange = true;


    while(anyChange)
    {
        anyChange = false;

        for(FaultPropagationModelNode node : fpm.getNodes())
        {
            // remove all nodes which have no parent
            if (node.getParent() == null)
            {
                fpm.removeNode(node);
                anyChange = true;
                continue;
            }

            // remove all nodes which are transitive and have no children
            if (node.isTransitive() && node.getChildren().length == 0)
            {
                fpm.removeNode(node);
                anyChange = true;
            }
        }
    }
}
```

A.4. Fault Localization: Timing Based Ranking

```
// returns ordered set of root cause candidates
public FaultPropagationModelMode[] timingRanking(
        FaultPropagationModel fpm,
        boolean ascendingOrdering)
{
    FaultPropagationModelMode[]        fpmm;


    // extract all root cause candidates from fpm
    fpmm = fpm.getAllLeafModes();

    // candidates are ordered based on distance of thier timestamp of
       symptom
    // from root node (client) timestamp
    fpmm = orderModes(fpm.getRoot().getTimestamp(), fpmm);

    return fpmm;
}
```

A.5. Fault Localization: Bayesian Network Based Ranking

```
// returns set of ordered root cause candidates with thier probability
public SortedMap<double, BNetNode> BNetRanking(FaultPropagationModel fpm)
{
    BNet        bnet;



    // load BNet from FPM
    bnet = createBNet(fpm);

    // create CPD
    createCPD(bnet);

    // assign evidence - the root node evidence = true
    bnet.getRoot().setEvidence(true);

    // run inference
    return runInference(bnet);
}

// isomorphic transformation of FPM to BNet
private BNet createBNet(FaultPropagationModel fpm)
{
    BNet        bnet = new BNet();

    createBNetNode(bnet, fpm.getRoot(), fpm.getRoot());
}

// recursive - create node in BNet and add children
private void createBNetNode(
        BNet bnet,
        FaultPropagationModelNode parent,
        FaultPropagationModelNode mode)
{
    bnet.addNode(mode.getId());
    bnet.addLink(mode.getId(), parent.getId());
    for(FaultPropagationModelNode child : mode.getChildren())
    {
        createBNetNode(bnet, mode, child);
    }
}

// create CPD for each node with parents
private void createCPD(BNet bnet)
{
    CPD          cpd;


    for(BNetNode node : bnet.getNodes())
    {
        // only nodes with parents have CPD
        if (node.getParents().lenght > 0)
        {
            // create noisy-OR CPD, with modifier 0.99 for each link
            cpd = calculateCPD(node.getParents(), 0.99);
            node.setCPD(cpd);
        }
    }
}

// inferes for each node its probability and returns sorted set of
   candidates
private SortedMap<double, BNetNode> runInference(BNet bnet)
{
    double                  pp;
    SortedMap               output = new TreeMap<double, BNetNode>();


    for (BNetNode node: bnet.getNodes())
    {
        // only for root cause candidates - leafs
        if (node.isLeaf())
        {
            // infer posterior probability
            pp = node.calculatePosteriorProbability();
            output.put(pp, node);
        }
    }

    return output;
}
```

A.6. Distributed Data Harvesting: Peer Selection

```
public String[] selectPeers(int peerCap, int hopCountLimit)
{
    String[]        candidates;
    String[]        peers;


    candidates = findCandidates(hopCountLimit);
    peers = choosePeers(candidates, peerCap);

    return peers;
}

// returns list of candidates
public String[] findCandidates(int hopCountLimit)
{
    ArrayList<String>               candidates = new ArrayList<>();
    RoutingTableRecord[]            rtrs;


    // retrives current records of routing table
    rtrs = getRoutingTableRecords();

    for(RoutingTableRecord rtr : rtrs)
    {
        // candidates must be within the distance limit
        if (rtr.metric <= hopCountLimit)
        {
            candidates.add(rtr.getIp());
        }
    }

    return candidates;
}

// choose randomly peers from candidates
public String[] choosePeers(String[] candidates, int peerCap)
{
    String[]                                peers;
    String                                    randomPeer;


    // if candidate set is bigger than the cap
    if (candidates.length > peerCap)
    {
        for(int i = 0; i < peerCap; i++)
        {
            randomPeer = randomlySelectPeer(candidates);
            peers.add(randomPeer);
        }
    }
    // otherwise use all available candidates
    else
    {
        peers = candidates;
    }

    return peers;
}
```

A.7. Distributed Data Harvesting: Dataset Calculation

```
// calculates dataset to be send to a peer
public Dataset calculateDataset(String peer, long maximumAgeOfDataToSynch)
{
    Dataset      ds = new Dataset();


    // for each time series stored
    for(Timeseries ts : getAllTimeseries())
    {
        // add new data of the time series
        addTimeseries(ds, ts, maximumAgeOfDataToSynch);
    }

    return dataset;
}

// add data of the time series to the dataset
private void addTimeseries(
        Dataset ds,
        Timeseries ts,
        String peer,
        long maximumAgeOfDataToSynch)
{
    Timeslot[]       timeslots;
    long             lastTimestamp;
    long             maxAge;
    long             oldestTimeslotToInclude;


    // get timestamp of the last successful synchronization with the peer
    lastTimestamp = getLastSuccessfulSynchTimestamp(peer, ts);

    // get the maximum age of tranferable data
    maxAge = Now() - maximumAgeOfDataToSynch;

    // by choosing the later of these two
    // select time since when time slots should be included
    // (data should not be send again and not older then max age)
    oldestTimeslotToInclude = maxAge > lastTimestamp ? maxAge :
        lastTimestamp;

    // extract time slots newer then
    timeslots = ts.getTimeslotsNewerThen(oldestTimeslotToInclude);

    // if there are not new data, do not include the time series
    if (timeslots.length == 0) return;

    // trim empty timeslots
    timeslots = trimTimeslots(timeslots);

    // add timeslots into the dataset
    ds.addTimeseriesData(ts.ID, timeslots);
}
```

A.8. Distributed Data Harvesting: Transfer of Dataset

```
// this method is an example of synchronous send approach
// in real implementation is required asynchronous and parallel approach
public void sendDatasetToPeer(String peer, Dataset ds, int timeout)
{
    Message         msg = new Message();
    PeerProxy       proxy = new PeerProxy(peer);
    boolean         success;


    // store dataset into a message
    msg.addDataset(ds);

    // send message with dataset and set response timeout
    success = proxy.sendMessage(msg, timeout);

    // message was received by peer and confirmation response arrived on
       time
    if (success)
    {
        // stores timestamp of latest transfered time slot
        // of each time series contained in the dataset
        storeLatestSuccessfulSynchTimestampa(ds, peer);
    }
}
```

Bibliography

[1] A. Adya, P. Bahl, R. Chandra, and L. Qiu. Architecture and techniques for diagnosing faults in IEEE 802.11 infrastructure networks. In *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, pages 30–44, New York, NY, USA, 2004. ACM.

[2] J. Ahrenholz. Comparison of core network emulation platforms. In *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, pages 166–171, 2010.

[3] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim. Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7, 2008.

[4] J. Ahrenholz, T. Goff, and B. Adamson. Integration of the core and emane network emulators. In *MILITARY COMMUNICATIONS CONFERENCE, 2011 - MILCOM 2011*, pages 1870–1875, 2011.

[5] P. Bahl, P. Barham, R. Black, R. Ch, M. Goldszmidt, R. Isaacs, S. K, L. Li, J. Maccormick, D. A. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering dependencies for network management. In *Fifth Workshop on Hot Topics in Networks*, Nov. 2006.

[6] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 13–24. ACM, August 2007.

[7] P. Barham, R. Black, M. Goldszmidt, R. Isaacs, J. MacCormick, R. Mortier, and A. Simma. Constellation: Atomated discovery of service and host dependencies in networked systems. Technical Report MSR-TR-2008-67, Microsoft Research, 2008.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004.

[9] S. Basu, F. Casati, and F. Daniel. Toward web service dependency discovery for SOA management. In *Proceedings of the IEEE International Conference on Services Computing*, pages 422–429. IEEE Computer Society, 2008.

[10] P. Bellavista, A. Corradi, and E. Magistretti. Comparing and evaluating lightweight solutions for replica dissemination and retrieval in dense manets. In *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, pages 43–50, 2005.

[11] P. Bellavista, A. Corradi, and E. Magistretti. Redman: A decentralized middleware solution for cooperative replication in dense manets. In *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*, PERCOMW '05, pages 158–162, Washington, DC, USA, 2005. IEEE Computer Society.

[12] K. Birman. The promise, and limitations, of gossip protocols. *SIGOPS Oper. Syst. Rev.*, 41(5):8–13, Oct. 2007.

[13] A. Bouloutas, S. Calo, and A. Finkel. Alarm correlation and fault identification in communication networks. *Communications, IEEE Transactions on*, 42(234):523–533, 1994.

[14] A. Cavalcante and M. Grajzer. Fault propagation model for ad hoc networks. In *Proceedings of the IEEE International Conference on Communications*, pages 1–5, June 2011.

[15] C. Chen, A. Zaidman, and H.-G. Gross. A framework-based runtime monitoring approach for service-oriented software systems. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, QASBA '11, pages 17–20, New York, NY, USA, 2011. ACM.

[16] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based faliure and evolution management. In *Proceedings of the Symposium on Networked Systems Design and Implementation*. USENIX Association, 2004.

[17] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 117–130. USENIX Association, 2008.

[18] S. Chessa and P. Santi. Comparison-based system-level fault diagnosis in ad hoc networks. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 257–266, 2001.

[19] D. Dechouniotis, X. Dimitropoulos, A. Kind, and S. Denazis. Dependency detection using a fuzzy engine. In *Proceedings of the 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, number 4785 in Lecture Notes in Computer Science, pages 110–121. Springer-Verlag, 2007.

[20] A. Dimakis, S. Kar, J. Moura, M. Rabbat, and A. Scaglione. Gossip algorithms for distributed signal processing. *Proceedings of the IEEE*, 98(11):1847–1864, 2010.

[21] G. Ding and B. Bhargava. Peer-to-peer file-sharing over mobile ad hoc networks. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 104–108, 2004.

[22] M. Elhadef, A. Boukerche, and H. Elkadiki. Diagnosing mobile ad-hoc networks: Two distributed comparison-based self-diagnosis protocols. In *Proceedings of the 4th ACM International Workshop on Mobility Management and Wireless Access*, pages 18–27, New York, NY, USA, 2006. ACM.

[23] M. Elhadef, A. Boukerche, and H. Elkadiki. A distributed fault identification protocol for wireless and mobile ad hoc networks. *Journal of Parallel and Distributed Computing*, 68(3):321–335, Mar. 2008.

[24] I. K. Eltahir. The impact of different radio propagation models for mobile ad hoc networks (MANET) in urban area environment. In *2nd International Conference on Wireless Broadband and Ultra Wideband Communications*. IEEE, Aug. 2007.

[25] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

[26] M. Fecko and M. Steinder. Combinatorial designs in multiple faults localization for battlefield networks. In *Proceedings of the IEEE Military Communications Conference*, volume 2, pages 938–942, 2001.

[27] S. Geyik, B. Szymanski, P. Zerfos, and D. Verma. Dynamic composition of services in sensor networks. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 242 –249, july 2010.

[28] T. Hara. Effective replica allocation in ad hoc networks for improving data accessibility. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1568–1576 vol.3, 2001.

[29] T. Hara. Replica allocation methods in ad hoc networks with data update. *Mob. Netw. Appl.*, 8(4):343–354, Aug. 2003.

[30] T. Hara, N. Murakami, and S. Nishio. Replica allocation for correlated data items in ad hoc sensor networks. *SIGMOD Rec.*, 33(1):38–43, Mar. 2004.

[31] M. Hasan, B. Sugla, and R. Viswanathan. A conceptual framework for network management event correlation and filtering systems. In *Integrated Network Management, 1999. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on*, pages 233–246, 1999.

[32] M. Hauspie, D. Simplot, and J. Carle. Replication decision algorithm based on link evaluation services in manet. In *CNRS UPRESA 8022ŮLIFL University Lille*, 2002.

[33] D. Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Proceedings of the Fifth Annual Conference on Uncertainty in Artificial Intelligence*, pages 163–172, 1989.

[34] T. Heer, S. Gotz, S. Rieche, and K. Wehrle. Adapting distributed hash tables for mobile ad hoc networks. In *Proceedings of the 4th annual IEEE international conference on Pervasive Computing and Communications Workshops*, PERCOMW '06, pages 173–, Washington, DC, USA, 2006. IEEE Computer Society.

[35] J.-L. Huang, M.-S. Chen, and W.-C. Peng. Exploring group mobility for replica data allocation in a mobile environment. In *Proceedings of the twelfth international*

conference on Information and knowledge management, CIKM '03, pages 161–168, New York, NY, USA, 2003. ACM.

[36] G. Karumanchi, S. Muralidharan, and R. Prakash. Information dissemination in partitionable mobile ad hoc networks. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, SRDS '99, pages 4–, Washington, DC, USA, 1999. IEEE Computer Society.

[37] I. Katzela and M. Schwartz. Schemes for fault identification in communication networks. *IEEE/ACM Trans. Netw.*, 3(6):753–764, Dec. 1995.

[38] A. Klemm, C. Lindemann, and O. Waldhorst. A special-purpose peer-to-peer file sharing system for mobile ad hoc networks. In *Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th*, volume 4, pages 2758–2763 Vol.4, 2003.

[39] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Proceedings of the fourth international symposium on Integrated network management IV*, pages 266–277, London, UK, UK, 1995. Chapman & Hall, Ltd.

[40] B. Ko, S. Liu, M. Zafer, H. Wong, and K. Lee. Gateway selection in hybrid wireless networks through cooperative probing. In *Proceedings of the IFIP/IEEE Integrated Network Management Symposium (IM)*, 2013.

[41] G. Kortuem, J. Schneider, D. Preuitt, T. Thompson, S. Fickas, and Z. Segall. When peer-to-peer comes face-to-face: collaborative peer-to-peer computing in mobile adhoc networks. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 75–91, 2001.

[42] J.-L. Kuo, C.-H. Shih, C.-Y. Ho, and Y.-C. Chen. A cross-layer approach for realtime multimedia streaming on wireless peer-to-peer ad hoc network. *Ad Hoc Netw.*, 11(1):339–354, Jan. 2013.

[43] J.-G. Lou, Q. Fu, Y. Wang, and J. Li. Mining dependency in distributed systems through unstructured logs analysis. *SIGOPS Operating Systems Review*, 44:91–96, March 2010.

[44] J. Luo, J.-P. Hubaux, and P. T. Eugster. Pan: providing reliable storage in mobile ad hoc networks with probabilistic quorum systems. In *Proceedings of the 4th ACM*

international symposium on Mobile ad hoc networking & computing, MobiHoc '03, pages 1–12, New York, NY, USA, 2003. ACM.

[45] S. Mo, J. Hsu, J. Gu, M. Luo, and R. Ghanadan. Network synchronization for distributed MANET. In *Military Communications Conference*. IEEE, Nov. 2008.

[46] A. Moon and H. Cho. Energy efficient replication extended database state machine in mobile ad hoc network. In *IADIS International Conference on Applied Computing*, pages 224–228, 2004.

[47] M. Natu and A. Sethi. Adaptive fault localization in mobile ad hoc battlefield networks. In *Proceedings of the IEEE Military Communications Conference*, pages 814–820, Oct. 2005.

[48] M. Natu and A. S. Sethi. Using temporal correlation for fault localization in dynamically changing networks. *International Journal of Network Management*, 18(4):301–314, Aug. 2008.

[49] L. B. Oliveira, I. G. Siqueira, D. F. Macedo, A. A. F. Loureiro, H. C. Wong, and J. M. Nogueira. Evaluation of peer-to-peer network content discovery techniques over mobile ad hoc networks. In *Proceedings of the Sixth IEEE International Symposium on World of Wireless Mobile and Multimedia Networks*, WOWMOM '05, pages 51–56, Washington, DC, USA, 2005. IEEE Computer Society.

[50] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 126–137, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[51] P. Padmanabhan, L. Gruenwald, A. Vallur, and M. Atiquzzaman. A survey of data replication techniques for mobile ad hoc network databases. *The VLDB Journal*, 17(5):1143–1164, 2008.

[52] L. Paradis and Q. Han. A survey of fault management in wireless sensor networks. *Journal of Network and Systems Management*, 15(2):171–190, June 2007.

[53] L. Popa, B.-G. Chun, I. Stoica, J. Chandrashekar, and N. Taft. Macroscope: End-point approach to networked application dependency discovery. In *Proceedings of the*

5th International Conference on Emerging Networking Experiments and Technologies, pages 229–240. ACM, 2009.

[54] L. Qiu, P. Bahl, A. Rao, and L. Zhou. Troubleshooting wireless mesh networks. *SIGCOMM Computing Communications Review*, 36(5):17–28, Oct. 2006.

[55] M. Rabbat and R. Nowak. Distributed optimization in sensor networks. In *Information Processing in Sensor Networks, 2004. IPSN 2004. Third International Symposium on*, pages 20–27, 2004.

[56] A. Shaheen and L. Gruenwald. Group based replication for mobile ad hoc databases (gbrmad). In *Technical Report, University of Oklahoma, Norman*, 2000.

[57] W. She, I.-L. Yen, and B. Thuraisingham. WS-Sim: A web service simulation toolset with realistic data support. *Computer Software and Applications Conference Workshops*, 0:109–114, 2010.

[58] M. Steinder and A. Sethi. End-to-end service failure diagnosis using belief networks. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 375–390, 2002.

[59] M. Steinder and A. S. Sethi. Probabilistic fault diagnosis in communication systems through incremental hypothesis updating. *Comput. Netw.*, 45(4):537–562, July 2004.

[60] M. Steinder and A. S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165–194, 2004.

[61] T. A. Stephenson. An introduction to Bayesian network theory and usage. Technical Report IDIAP-RR-03-2000, Dalle Molle Institute for Perceptual Artificial Intelligence, Martigny, Switzerland, Feb. 2000.

[62] S. Tati, P. Novotny, B. J. Ko, A. Wolf, A. Swami, and T. La Porta. Diagnosing degradation of services in hybrid wireless tactical networks. In *SPIE Defense, Security, and Sensing*, pages 874210–874210. International Society for Optics and Photonics, 2013.

[63] N. Tcholtchev, M. Grajzer, and B. Vidalenc. Towards a unified architecture for resilience, survivability and autonomic fault-management for self-managing networks.

In *Proceedings of the International Conference on Service-Oriented Computing Workshops*, number 6275 in Lecture Notes in Computer Science, pages 335–344. Springer, 2010.

[64] J. Vomlel. Exploiting functional dependence in bayesian network inference. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, UAI'02, pages 528–535, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[65] K. Wang and B. Li. Efficient and guaranteed service coverage in partitionable mobile ad-hoc networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 1089–1098 vol.2, 2002.

[66] S. Wang and M. A. M. Capretz. A dependency impact analysis model for web services evolution. In *Proceedings of the IEEE International Conference on Web Services*, pages 359–365. IEEE Computer Society, 2009.

[67] Y. Yu, B. Krishnamachari, and V. Prasanna. Energy-latency tradeoffs for data gathering in wireless sensor networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 1, pages –255, 2004.

[68] D. Zhou and T.-H. Lai. An accurate and scalable clock synchronization protocol for IEEE 802.11-based multihop ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(12):1797–1808, Dec. 2007.