A Framework for Automatic Reconfigurations of Protocol Stacks in Ubiquitous Computing Systems

Mahdi Niamanesh, Rasool Jalili
Department of Computer Engineering
Sharif University of Technology, Tehran, Iran
niamanesh@mehr.sharif.edu, jalili@sharif.edu

Abstract

Ubiquitous computing environment includes wireless networks, autonomic networked systems and devices with heterogeneous standards and protocols for different contexts and situations. Autonomic networked systems require mechanisms for automatic on-the-fly reconfigurations in their protocol stacks to be able to operate in different situations and contexts. Automatic reconfigurations allow the system to continue its normal operation without any need for human intervention. Since every protocol in a protocol stack has at least one peer protocol in another system, a dynamic reconfiguration of a protocol may raise the need for a reconfiguration in the peer stack. Depending on the reconfiguration type, the reconfiguration maybe carried out in a single or in both peer systems. For an assured dynamic (run-time) reconfiguration of a protocol, the execution of the protocol component is stopped in a safe state, a new protocol is initialized, and stacks switch to the new protocol at the same time. This paper proposes a protocol stack framework to support automatic and assured reconfigurations of protocol components. Our evaluation and performance measurement show the feasibility of automatic reconfigurations and an acceptable level of transparency in dynamic reconfigurations.

Keywords: Protocol Stack, Automatic Reconfigurations, Dynamic Reconfiguration
1 Introduction

Ubiquitous computing environment includes wireless networks, autonomic networked systems and devices with heterogeneous standards and protocols for different contexts and situations [20]. The Software Radio [3, 8] offers dynamic reconfigurability for protocol stacks of such systems and devices in order to facilitate applications such as changing routing algorithms of switches, changing security modules in protocol stacks, bug fixing, and customizing the protocol stack of a device for better performance.

In general, in a software system, dynamic reconfiguration of a component to a new one includes such phases as freeze (stopping the current execution of the component), change (adding/binding a new component and unbinding/removing the unnecessary old component from the system), state transfer (finding and initializing a proper state in the new component in order to resume the execution), and re-execution (resuming the execution from a non-initial state in the new component) [15]. In order to have assured reconfiguration, the old component should be frozen in a “safe state” and the new component should resume the execution from a “reachable” state [6].

In the context of protocol stack reconfigurations, since each protocol is defined at least between two peer components, reconfiguration of a running protocol component may require a corresponding reconfiguration in the peer component(s). For example, consider a reconfiguration that changes TCP component in a TCP/IP protocol stack into SCTP component [22]. Both TCP and SCTP have the same role, as transport protocol. For this reconfiguration, both peers should be synchronized in terms of the change (e.g., from TCP to SCTP) and freeze state. However, in reconfiguring TCP component into TCP-Nice component [24], since TCP-Nice is backward-compatible with TCP, the reconfiguration can be performed in a sender stack transparently and independent of its peer stack. As a result, reconfigurations can be categorized into single reconfigurations (involving a single stack) and distributed reconfigurations (involving changes in two or more stacks).

Autonomic systems in ubiquitous computing environment should have dynamic protocol stacks that can automatically detect reconfiguration types and perform the assured reconfigurations. Related work for dynamic protocol stacks only address single or distributed reconfigurations. For example [9, 11] suppose a running protocol stack as a stand-alone system (not in a network) and perform a single reconfiguration. In order to provide an assured reconfiguration, they have defined the safe state of a running component as a state where the component has no data and is not in any interaction with the other components [7, 11]. In the Software Radio and Cognitive Radio domains, related work has mainly concentrated on device protocol stack reconfigurations [8] and cognitive engine development for software radio (e.g., [14]). In the Internet domain, related work such as [19, 23] have concentrated on changing the communication protocol between two systems. These works limit safe states to states in which the old protocol execution has been completed and the new protocol should start a new execution from an initial state.

In this paper, the reconfiguration problem is defined as changing the protocol between two peer protocol components at run-time automatically. The reconfiguration maybe performed in a single peer (single reconfiguration) or in both peers (distributed reconfiguration) Our idea is to keep enough information about protocols to perform automatic and assured reconfigurations. Based on our knowledge about two communicating protocol components, the reconfiguration type is detected, and is carried out in a safe state. We do not limit safe states to the end of protocol executions. For such reconfigurations, we propose a software framework including mechanisms for dynamic reconfiguration management and control for both types of reconfiguration. Through the framework, each stack can request a reconfiguration from its peer while they are communicating. Our framework for automatic reconfigurations guarantees smooth change of the old protocol to the new one autonomously.

The rest of this paper is organized as follows. Section 2 describes background about protocol executions, protocol reconfigurations, and reconfiguration assurance in protocol stacks. In Section 3, we explain the proposed framework. Mechanisms for reconfiguration management and control for single and distributed reconfigurations are presented. In Section 4, we describe implementation and evaluation of the proposed framework. In Section 5, we discuss related work and Section 6 concludes the paper.

2 Background

In this section, we describe a simple model for protocol execution. Afterwards, two types of protocol stack reconfigurations are explained and finally, we state assurance in reconfigurations of two communicating stacks.

2.1 Protocol Execution

We consider a simple model for describing the reconfiguration problem in communicating protocol stacks.
Figure 1 shows two communicating protocol stacks, Stack1 and Stack2. In each layer of the stacks, one component, which we refer to as protocol component, implements functionality of its corresponding protocol. For instance, in the figure, components P1 and P2 provide functionality of protocol P. A protocol component is a set of rules as well as a processing entity that governs communication between communicating peers. The protocol components interact with each other by exchanging protocol messages.

State of a protocol component describes all useful information about the protocol in a point of execution. Such information includes values of all variables and contents of all input/output buffers, related to the component. For simplicity, we split states of a protocol component into macro-states and micro-states. Macro-states describe states of the protocols’ finite state machines. Examples of macro-states in the TCP protocol are CLOSED and ESTABLISHED. Micro-states describe states of protocols at run-time, to maintain information for operations such as reliability, error handling, and congestion control. We use macro-states of components in specification levels and micro-states in execution levels.

2.2 Reconfigurations in Protocol Stacks

With two communicating protocol stacks, a dynamic reconfiguration in a component may lead to a corresponding reconfiguration in its peer component. In our perspective, protocol stack reconfigurations are categorized into two types, single reconfiguration and distributed reconfiguration. Figure 2 depicts scenarios for single and distributed reconfigurations. Figure 2-(a) depicts a situation in which in one of the stacks (e.g., Stack1), a protocol changes into another variation that is supported by the peer protocol component (in Stack2). According to the figure, the old protocol component, P1, is replaced by a new component P3, which can “safely” interoperate with the old peer component, P2, in the peer stack. In this case, the new component, P3, is “backward-compatible” with the old one. Therefore, the reconfiguration can be carried out in the changing stack without disrupting the peer stack.

Figure 2-(b) depicts a scenario in which two peer components in the two stacks are synchronously replaced with new protocol components. For two communicating TCP/IP protocol stacks, changing TCP protocol of two stacks into SCTP protocol [22] can be an example of this type.

For single reconfigurations, the goal is to reconfigure a running protocol component independent of its peer, transparently (in point of its peer component view). However, the goal of distributed reconfigurations is to reconfigure two peer components synchronously.

2.3 Reconfiguration Assurance

Intuitively, a dynamic reconfiguration that changes a running component into a new one is assured if after the changing phase, the new component can be executed just as if it has been executed from its initial state [6]. Based on this notion, if the new component
starts re-execution from a reachable state\textsuperscript{1}, then the reconfiguration will be assured. Based on \cite{6}, to achieve such a reachable state, the running component should be frozen in a safe state, and its state should be transferred into the new component. Considering two communicating components as a distributed component, its state is the global state\textsuperscript{2} of the two components and the reconfiguration of two communicating components is assured if after the reconfiguration, the execution resumes from a reachable global state.

We explain requirements for assured reconfigurations (both single and distributed reconfigurations) in the following:

**Safe state** A safe state for a reconfiguration has been defined as a state that has no interaction with the other components \cite{7, 11}. A component in a safe state does not accept new requests, does not initiate new operations, and all its initiated operations have been completed \cite{18}. However, for two communicating stacks, the global state should be safe. Accordingly, two communicating components should be frozen in a safe global state, and two new components should resume the execution from a reachable global state.

**State transfer** Execution of the new component maybe resumed from a non-initial state, which we refer to as the restarting state. The old component’s state should be transferred to the restarting state. In a reconfiguration of two peer components, the restarting states of both peers should be initialized. An important point in the state transfer is the possible dependency of the states of two peers upon each other. For example, a UDP sender component should know the port number of the peer UDP receiver. For the state transfer, firstly, it is necessary to find the restarting state in the new component; secondly, the new restarting state should be initialized to resume the execution; thirdly, some parts of the new component’s state may require to be initialized based on its peer component.

### 3 DRAPS Framework

DRAPS (Dynamic-Reconfigurable Architecture for Protocol Stack) is an extendable framework that presents assured and synchronous dynamic reconfiguration for two communicating protocol stacks. Without losing the generality, we assume that each protocol in DRAPS, is implemented as a distinct component. A protocol component is a set of rules as well as a processing entity that governs the communication between communicating peers.

Architectural components of DRAPS are shown in Figure 3. As shown, DRAPS is built out from a core framework and some plug-in components. The core framework is responsible to perform assured dynamic reconfigurations and consists of two components, namely, Reconfiguration Management and Control (RMC), and Protocol Knowledge Base (PKB). RMC provides mechanisms for automatic reconfiguration management of two peer stacks. PKB is a knowledge base component, responsible for initializing protocols specifically when they are resumed from non-initial states.

Plug-in components in DRAPS present extendability for the framework by providing supplementary requirements of dynamic reconfigurations. For instance, a repository of reconfigurable components is not an essential requirement in a reconfiguration; however, it is a supplementary requirement in every reconfiguration framework. DRAPS provides a plug-in type for such a component repository.

![Figure 3. Architecture of DRAPS including core framework and plug-ins](image)

#### 3.1 The Protocol Knowledge Base

We introduce a knowledge base component, called Protocol Knowledge Base (PKB), to support assured and dynamic reconfigurations for protocol stacks of autonomic systems. Theses systems should detect reconfiguration types and perform assured reconfigurations without any need for human intervention. For this reason, we keep a protocol knowledge in the PKB component. Specifically, PKB contains three types of knowledge, namely, peer-compatibility of protocols,
role-compatibility of protocols, and protocol state information (Figure 4) all explained in the following.

![Figure 4. PKB consists of 1) Knowledge of role-compatibility of protocols, 2) Knowledge of peer-compatibility of protocols, and 3) Knowledge of protocols states](image)

### 3.1.1 Peer-Compatible Protocols

Every protocol reconfiguration requires commitment of its endpoint components on a change and a freeze state. Commitment on the change means the two components change into compatible protocols. For example, when a peer component changes into the TCP protocol, the other peer can not choose UDP protocol for its transport protocol. For this reason, we define peer-compatibility relationship between two protocol components as follows. Two components have peer-compatibility relationship (are peer-compatible) if they can inter-operate with each other as two peer components. We introduce two types of peer-compatibility, i.e., **strongly peer-compatible** if they can inter-operate in such a way that all their functionalities can be potentially used in the communication, **weakly peer-compatible** if parts of functionalities of the components can not be used, due to problems such as version mismatching. For example, TCP and TCP-NewReno [4] (or TCP-SACK [10]) components, as two communicating peers, have the weak peer-compatibility, whereas TCP and TCP-Nice possess strong peer-compatibility [17]. The PKB component keeps all types of peer-compatible protocols.

The commitment on the freeze state necessitates two peer components to start the reconfiguration in a safe global state. For this purpose, the PKB component keeps all peer-compatible macro-states for each pair of peer-compatible protocols. By peer-compatible macro-states, we mean two “compatible” macro-states from two peer-compatible protocols\(^3\). For example, when a TCP sender component is in the ESTABLISHED state, its peer component, which is a TCP receiver, can not be in the CLOSED state.

### 3.1.2 Role-Compatible Protocols

In a reconfiguration of an existing component into a new component, the new one should play the same role as the existing one. For example, we may reconfigure TCP into SCTP or UDP, but it is not possible to reconfigure TCP into IP. In fact, the role of TCP and IP differs completely. We define role-compatibility relationship between two protocol components that can be reconfigured to each other. A pair of protocol and its backward-compatible version is a well-known example of role-compatible protocols. We define three types of role-compatibility relationship between two protocol components, which are determined based on the peer-compatibility of new component and the peer component. Considering a component, \(P_1\), and its peer, \(P_2\), which are strongly peer-compatible components, we describe types of role-compatibility as follows.

We call a component, \(P_3\), as **fully backward-compatible** (FBC, in short) with \(P_1\), if \(P_3\) can be strongly peer-compatible with \(P_2\). For two communicating peer components, we can perform a single reconfiguration by changing a component to one of its FBC components. For a protocol component, its FBC protocols are those that inherently require changes to only one endpoint component. For example, both the Fast Recovery modification to the sender-side TCP [1] and TCP-Nice are transparent to the receiver.

**Not Backward-Compatible** role-compatibility (NBC, in short) relationship is held between \(P_1\) and \(P_3\), if \(P_3\) can not have any peer-compatibility with \(P_2\). In this case, the reconfiguration necessitates a distributed reconfiguration. For example, SCTP can be regarded as a role-compatible protocol with TCP and UDP. But changing SCTP into TCP or UDP should be synchronously carried out in both peer components to be of value.

**Partially Backward-Compatible** role-compatibility (PBC, in short) relationship is hold between \(P_1\) and \(P_3\), if \(P_3\) can have weak peer-compatibility with \(P_2\). In this case, any reconfiguration from \(P_1\) into \(P_3\) can be carried out either single or distributed. For a protocol, PBC protocols are those that have the potential to be either more effective if both peer components could be reconfigured and the new functionality could be oper-\(^3\)In [16], we have formally defined two compatible states. We have proposed protocol automata specifying the communication of two communicating components. Each state of protocol automaton is composed of two compatible states from the two components.
tional between the sender and receiver. If we reconfigure only one peer, we will have partial or no new value in the communication. For example, for two communicating TCP components, changing one of them into TCP-NewReno, which uses a heuristic interpretation of duplicate acknowledgments to avoid timeouts, does prove useful. However, by changing the peer component into TCP-NewReno, the communication will have more useful. Through changing one of the components into TCP-SACK, which provides better recovery from losses, the communication will serve no benefit unless the peer component is TCP-SACK enabled or we reconfigure the peer component into TCP-SACK as well. In these cases, we can perform either a single or distributed reconfiguration.

For each pair of role-compatible protocols, PKB contains their role-compatibility type. Moreover, it contains role-compatible macro-states of the two protocols. A pair of role-compatible macro-states of two role-compatible protocols consists of two corresponding states that can play the same role in the two protocols. This helps reconstructions as follows: when a protocol stops in a state (freeze state), a corresponding state (restarting state) is found through its role-compatible macro-states to resume the execution.

3.1.3 Protocol State Information

All required knowledge for finding safe states and initializing restarting states is kept in PKB.

We present a model for state representation of protocol components by introducing protocol control block (PCB) that contains state information. This information includes addressing parameters, buffers (sent, received, un-acknowledged, etc.), protocol segment variables, counters, timers, send variables, receive variables, and the other required parameters. Protocol developers should implement one PCB data structure for each protocol component.

PCB handlers are used to set values of PCB parameters during a state transfer. They are also used in distributed reconstructions to set values of the remote PCBs in peer systems. In this case, we call them remote PCB handlers. As an example of the application of remote PCB handlers, consider a reconfiguration in UDP protocol. In UDP protocol, the UDP sender should know the port number of the UDP receiver; therefore, the receiver should send its port number to the sender. PKB contains PCB parameters and formats for different protocols.

In the state transfer, the old PCB is used to valuate the same parameters in the new PCB. Then, the PCB handlers are used to complete the state transfer and finally the remote PCB handlers from the peer (only in distributed reconstructions) are applied.

3.2 Reconfiguration supports for Protocol Components

Reconfigurable protocol components should provide some extra functionalities to support dynamic reconstructions. For each protocol component, protocol developers should implement reconfiguration interface including saveState() and restoreState() methods for state transfer, start() and stop() methods to start and stop execution of the component. Finally, the semiFreeze() method should be implemented for each component in order to support freezing the component in a proper state.

Moreover, dynamic reconstructions of protocol components require indirect communication between two adjacent protocol components (layers) in order to present transparent run-time reconstructions [5]. For this reason, a wrapper component is used between two adjacent protocol components. For a protocol component, the wrapper presents interface of the protocol component and manages the incoming requests to the component, by buffering them during the freeze periods.

3.2.1 Finding a Safe State

Due to the independence of protocol stack components (layers), finding a safe state for a dynamic reconfiguration can be carried out in a simple way. We define a safe reconfiguration point (SRP) as a point of execution of a protocol component where the component is at the beginning or at the end of the processing of an input packet. In other words, the states just after writing a packet into the output buffer or before reading a packet from the input buffer are SRPs. In such states, either no operation is started on the packet or all the operations are completed. Usually, macro states of protocols are like this and we use them as SRPs. This definition is consistent with [11], where all operations of the component in SRP should be completed.

Generally, finding a SRP in an execution of a component is a reachability problem, which is undecidable [6]. To cope with this problem, we ask protocol developers to “mark” some SRPs (macro states) in the source code of protocols. Therefore, protocol developers are asked to put some pieces of code in the source code of protocol components to enable them to report macro-states during the execution if requested. In a normal execution mode, a component does not report

4Based on [16], we define two role-compatible states in two substitutable components as two bisimilar states.
SRPs. However, we introduce a semi-freeze mode in which the component reports every SRP (like in debug mode) upon reaching its execution. The RMC component monitors the reported states of the component in the semi-freeze mode and stops its execution upon reaching the proper SRP for a dynamic reconfiguration.

For distributed reconfigurations, the global state of two peer components should be safe. A global state \((s, r)\) for two communicating components is a safe global state if \(s\) and \(r\) are locally SRP and their role-compatible states, denoted as \(s'\) and \(r'\) respectively, are peer-compatible with each other. Figure 5 depicts the scenario of finding a safe global state for a distributed reconfiguration. For each SRP state, say \(s\), RMC checks whether its role-compatible state in the new component, e.g., \(s'\), has a peer-compatible state in the new peer component. If so, the pair of reported state and its peer-compatible state in the peer component is the safe global state for the reconfiguration.

![Figure 5. The sequence of finding a safe global state](image)

### 3.3 Automatic Reconfiguration Management and Control

We offer automatic reconfiguration management and control (RMC) through a software component (RMC component) for protocol stacks. The RMC component can receive reconfiguration commands from different sources including system administrators, monitoring components in the system, or peer system RMCs. A reconfiguration command asks replacing an old protocol component with a new one.

We provide automatic reconfigurations in three phases; in the reconfiguration preparation phase, conditions for an assured reconfiguration are checked. In reconfiguration synchronization, two peer stacks synchronize on a dynamic reconfiguration. In reconfiguration execution the new component is installed and executed. In the following subsections, we explain automatic reconfiguration phases.

#### 3.3.1 Reconfiguration Preparation

Considering two communicating protocol stacks, we would like to reconfigure one of the stacks by sending a reconfiguration command indicating the currently running protocol component (old component) and a new component. We describe how RMC and PKB components can manage an automatic reconfiguration for these protocol stacks. Each protocol stack contains its own RMC and PKB components. Moreover, we have the following assumptions to simplify the problem:

- The communication channel between the two components is FIFO (First-In, First-Out), error-free, and having bounded communication delay.
- There is only one reconfiguration at a time. No concurrent reconfigurations are started.

The reconfiguration preparation is started upon receiving a reconfiguration command by a RMC. The goal is to check preconditions for assured reconfigurations. Therefore, RMC checks SRPs for the currently running component. The component should have at least one SRP and its role-compatible states, with respect to the reconfiguration, in the PKB component. Moreover, RMC checks the existence of new component’s PCB and PCB handlers.

#### 3.3.2 Reconfiguration Synchronization

The goal of reconfiguration synchronization is to determine a compatible reconfiguration for two peer stacks. Thus, the initiator RMC detects the received reconfiguration type, as depicted in Figure 6. If the role-compatibility between the old and new components is FBC then the reconfiguration type is single, and is carried out in the initiator stack. If the role-compatibility is PBC and the peer-compatibility of the new component and the peer component is strong then, the reconfiguration will be single in the initiator stack. Otherwise, the reconfiguration can be either single or distributed.

Three types of messages (START, RECON, and EoREC) are exchanged by two RMCs to determine and perform a compatible reconfiguration. The START message is a command for reconfiguration start and indicates the current (old) component and a set of all possible options for the peer component. Each option indicates an alternative component and a proper freeze
state for the peer. Alternative components are determined through the peer-compatibility relation with the new component in the initiator stack. The freeze state is determined based on safe global state. In addition, the START message includes a list of PCB values for remote PCB initialization. During the synchronization, the initiator RMC sends a START message to the peer RMC. The peer RMC, based on the received options, prepares a corresponding START message (including its running protocol name and options for the peer) and sends back to the initiator RMC. Now, the initiator RMC can decide on the reconfiguration type.

For distributed reconfigurations, the initiator RMC selects two peer-compatible protocols for the initiator and the peer stacks. The peer’s reconfiguration is determined in a RECON (RECONFIGuration) message. Each RMC sends the EoREC (End of RECONFIGuration) message to its peer RMC after finishing a reconfiguration. It only indicates that the reconfiguration has been successfully finished and that the new protocol is operational. Therefore, the EoREC message does not play any critical role in the reconfiguration of the two peers. The sequence of exchanging messages between two stacks during a synchronization is depicted in Figure 7. Note that RECON and EoREC messages are not used in single reconfigurations.

An important point in dynamic reconfigurations of two communicating protocol stacks is that, every reconfiguration causes communication inside the stacks to become limited or even blocked. For a synchronous reconfiguration of two peer stacks, this can lead to the problem in communication of their RMCs during the synchronization. However, as explained, the RMC components in the two stacks are synchronized in such a way that peer RMCs do not require any communication in the blocked period.

For single reconfigurations, the initiator RMC starts the reconfiguration execution, as described in the next subsection.

### 3.3.3 Reconfiguration Execution

The reconfiguration execution includes installation, freeze, and re-execution steps. Figure 8 depicts the steps for a component replacement in the reconfiguration execution phase. In the case of satisfaction of all preconditions for an assured reconfiguration, the RMC component installs the new component (Figure 8-(b)) and then, invokes the semiFreeze() method of the currently running component to start its semi-freeze mode (Figure 8-(c)). In this mode, RMC monitors the component execution and freezes it by invoking its stop() method in a SRP determined in the RECON
message. Afterwards, the RMC uses `saveState()` and `restoreState()` methods to transfer the state values in the old component’s PCB into the new component’s PCB (Figure 8-(d)). Later, the new component is re-executed through its `start()` method (Figure 8-(e)). Now the new component is available for its user components and the application can send data through the new protocol.

![Figure 8. Component replacement in the re-configuration execution](image)

It is worth noting that when two peers enter into the freeze mode, there may be some packets inside the communication channel (packets that have been sent by a component but not delivered to the peer component yet). In this case, although the macro-states of two peers are compatible, their micro-states (run-time states) are not consistent. If the old protocol is reliable, after changing into another reliable protocol, the new one can resend these packets. If the old or new protocol is unreliable, these packets may be lost in the communication.

### 4 Implementation and Evaluation

In this section, we evaluate the proposed framework for automatic dynamic reconfigurations. Overhead of using wrappers in protocol stack performance has been evaluated in related work such as [5, 19].

Java is chosen as the programming language due to its platform independence. To load and reconfigure protocol components at run-time, we use dynamic class loading. The configuration of the experimental environment includes two Centrino 1.5 and 1.7 GHz IBM personal computers with 256 and 512 MB memory. Linux (Debian 3.1 distribution) is used as the operating system.

Two communicating peer protocol stacks are considered, *Stack_1* and *Stack_2* (Figure 9). Both stacks are based on the DRAPS framework. Applications on the top of stacks exchange data with each other. Both applications use a light-weight version of TCP protocol, which we have implemented for the transport layer. The IP layer is simulated using two Linux FIFOs, one for outgoing data and the other for incoming data. Wrappers for TCP and IP layers have also been implemented.

In our experiments, reconfiguration commands are sent to the RMC in *Stack_1* to initiate the reconfigurations. Each stack contains 5 role-compatible components with the running TCP, namely, TCP-CA (TCP with Congestion Avoidance), SecureTCP (TCP with encryption capability), UDP, TCP-Nice, TCP-SACK. They have been implemented such that all their macro-states are SRPs. We have tested scenarios for FBC, PBC, and NBC role-compatibilities to demonstrate all cases of reconfiguration type detection.

Figure 10 shows duration of reconfiguration phases. Numbers in the chart are averaged over a large number of iterations. The preparation phase takes 71 milliseconds; the synchronization phase, which includes the times for sending and receiving START and RECON messages, takes 112 milliseconds; it depends on the round trip time. The execution phase takes 56 milliseconds. The maximum total time for an automatic reconfiguration takes 239 milliseconds in average.

![Figure 9. The experimentation environment](image)

Figure 11 shows the performance of execution phase of TCP (with empty buffer) reconfigurations according to the steps explained in the reconfiguration execution. The measured times for component installation, PKB operations, semi-freeze, and freeze are shown in
Figure 10. Time spent in preparation, synchronization, and execution phases

Figure 11. Time spent in execution

PKB operations include the time for finding SRPs and the peer-compatible macro-state. In addition, it includes the time to verify assurance of the reconfiguration through examining the possibility of complete initialization of new PCB. In our experiments, it takes around 3 milliseconds. According to the definition of SRP, the semi-freeze time depends on the processing time for input/output data in the TCP component. For the implemented TCP, in which all macro-states are SRPs, the semi-freeze time is 3 milliseconds.

The freeze time includes the time for state transfer, which takes up 17 milliseconds (5 milliseconds for copying the old PCB into the new one, and 12 milliseconds for invoking PCB-handlers). It is important to note that, the freeze time is the only period in which both old and new protocol components are unavailable. The total reconfiguration execution time, which includes installation, PKB operations, semi-freeze, and freeze times, is 56 milliseconds in DRAPS.

Table 1 shows a comparison of DRAPS and two related frameworks, which are implemented and evaluated in a similar environment with DRAPS. To be comparable, we do not include preparation time, in which we check preconditions for an assured reconfiguration in calculating the time for a single or distributed reconfiguration. In DRAPS, the single reconfiguration time is lower than the others because of using PCB and PCB handlers in state transfer. In [9], for state transfer, authors use Java serialization technique [12], which takes around 190 milliseconds. In DRAPS, distributed reconfigurations takes up 112 milliseconds for stacks synchronization and reconfiguration type detection. Whereas, DPF [2] does not support reconfiguration type detection.

Table 1. Comparison of reconfiguration times in the related frameworks ("-" = Not Supported)

<table>
<thead>
<tr>
<th>Framework</th>
<th>Time (ms)</th>
<th>Single</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPF [2]</td>
<td>77</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>Yueh-Feng Lee et. al. [9]</td>
<td>214</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>DRAPS</td>
<td>56</td>
<td>168</td>
<td></td>
</tr>
</tbody>
</table>

For the TCP protocol, since the timeout of TCP sockets is usually more than ten seconds on different machines, the reconfiguration is performed transparently from the application point of view. In general, it is important to demonstrate that, our framework can transparently reconfigure other protocols (besides TCP) as well. Therefore, the duration of the freeze period in a protocol should be less than the timeout of the protocol users (generally the upper layer protocol users). As the timeout value for most of the communication protocols is much more than a second, we can expect that the presented framework can transparently reconfigure other protocols as well. In our experiments for the TCP protocol, the freeze period takes 5 milliseconds for the default size of “send buffer”, which is much less than the TCP socket timeout in the application layer.

5 Related Work

Ioana Sora et al. propose the “protocol building block” description and protocol selection algorithms in [21]. They provide automatic stack composition through an algorithm to select building blocks in case all specified features are provided and all dependencies of selected components are satisfied. This paper offers protocol stack compositions at deployment-time.

In [13], a framework called OPtIMA is introduced for protocol stacks of software-radio based systems. The goal is to reconfigure protocol stacks built within the framework. OPtIMA presents the definition and provision of a library of classes, which can be used to build reconfigurable protocol stacks. However, the
framework does not support dynamic reconfiguration of protocols. It only presents customizibility of protocol stacks at deployment-time.

The DiPS/CuPS framework [11] is intended for the development of customizable system software. This research mainly concentrates on a component model for protocol components. Reflection points in DiPS/CuPS, which perform packet switching within a protocol stack, can only support run-time reconfigurations. Safe states for reconfigurations are defined based on protocol transactions; when the protocol execution is completed and buffers are empty, the state is safe for a reconfiguration. Specifically, DiPS/CuPS presents dynamic reconfigurations for protocol stacks with idle protocols. It does not provide flexible enough mechanisms for reconfigurations of running protocols (reconfiguration during protocol transactions).

Dynamic Protocol Framework (DPF) [2] mainly concentrates on building an adaptive protocol stack by automatic discovery and selection of protocol components. It supports on-the-fly reconfigurations of protocol stacks and provides a synchronization mechanism to ensure the compatibility of protocol stacks on communication peers. In DPF, safe states for reconfigurations are restricted to be at the end of protocol transactions. Moreover, supporting mechanisms for state transfer are not clearly defined and addressed. While the automation in DPF helps to select proper protocol components for a stack reconfiguration, DRAPS detects the reconfiguration types automatically. Based on the provided services of protocol components, DPF selects and composes a proper protocol stack. In contrast, DRAPS uses PKB to select proper protocol components. Moreover, through the synchronization, DRAPS detects the reconfiguration type and proposes proper components for the peer stack as well. Whereas DPF does not support protocol selection for the peer stack.

In [9], authors propose a Java-based framework that allows programmers to create, remove, and replace protocol modules at run-time. Programmers implement their components using the component framework. The framework can dynamically reconfigure components in safe states. To find a safe state, the framework uses lock management techniques to access a protocol module. When a reconfiguration command is received, the framework attempts to access the “write” lock to start the reconfiguration. Lock management adds complexity in programming protocol modules and also in the application layer. In contrast, we have implemented lock management in wrappers and so there is no extra overhead for the protocol programmer. For the state transfer, authors use the Java synchronization technique as a direct state transfer mechanism. However, we have used the PCB model for the indirect state transfer that causes a short freeze time.

In the context of distributed reconfiguration of communicating peers, there are quite a number of implementations that support distributed reconfigurations. In [19], authors propose an algorithm and a model for distributed reconfigurations of peers. A replacement module is responsible for reconfiguration control and management. It provides indirect access to the changing module, like the wrappers in DRAPS. In this paper, authors identify two properties of dynamically updateable systems, i.e., stack well-formedness and protocol operationability. Preserving these properties during a dynamic update guarantees the transparent updates. An algorithm, which can switch between different distributed agreement protocols (e.g., consensus and atomic broadcast), is proposed in that paper. As safe states are at the end of protocol transactions, they provide no mechanisms for finding global states or state transfer.

Maestro [23] supports only the replacement of whole protocol stacks; that is, in order to replace a protocol component, the whole stack containing the component has to be replaced. They propose a stack switching module installed on each machine to dynamically replace stacks. The main role of the module is to synchronize the start of the new stack. Protocols are developed within the Ensemble framework. They are decomposed into micro-protocols, each specialized to do a specific task. Since Maestro performs reconfigurations for the whole stack, the applications on top of the stack are blocked. In Maestro, a dynamic reconfiguration is started when the transaction of the old protocol has been completed and no data is exchanged between peers.

In DRAPS, dynamic reconfigurations can take place during the protocol transactions. Therefore, for long running servers DRAPS does not wait until the end of protocols executions. Distributed reconfigurations in DRAPS require freezing of both peer components in a safe global state; however, in related frameworks reconfigurations take place at the end of protocols transactions and there is no need for safe global state. The shortening of the freeze time in DRAPS is achieved through the PCB model, which incurs one-time development overhead for each protocol and presents a necessarily short time for the freeze period.

6 Conclusion

Ubiquitous computing environment includes autonomic networked systems requiring automatic dynamic reconfigurations in their protocol stacks. This paper
proposes a framework for automatic reconfigurations of protocol stacks. Although dynamic reconfiguration is not new, the proposed framework is novel in that it supports automatic reconfigurations of protocol stacks. We introduce two types of relationship between protocols, role-compatibility and peer-compatibility. Based on these relationships, the type of reconfiguration is detected and an assured reconfiguration is performed without the need for human intervention. Test scenarios for automatic single and distributed reconfigurations of TCP protocol in TCP/IP protocol stack are realized through the framework. The experimental results show that an acceptable transparency can be maintained using the proposed framework.

References