A Solution for Backward-Compatible Reconfigurations of Running Protocol Components in Protocol Stacks

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

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

Abstract

Forthcoming networked systems require mechanisms for on-the-fly reconfigurations in their protocol stacks to be able to operate in different situations and networks. Backward-compatible reconfigurations of protocols are fast and easy ways for new protocol distributions. However, performing such reconfigurations at run-time and for running protocol components, without disrupting peer components, is more desirable. This paper proposes a solution for dynamic reconfiguration management that can transparently reconfigure running protocol components in the middle of their protocol transaction. Mechanisms for the reconfiguration management including finding safe states as well as state transfer are proposed. For demonstration, we have implemented a prototype of the solution to reconfigure a running TCP component. Our experimental results on dynamic reconfigurations show that an acceptable transparency (through providing a short time for the freeze period) can be maintained using the proposed solution.
1 Introduction

Future communication and computation world, known as pervasive computing environment, includes wireless networks, networked systems and devices with heterogeneous standards and protocols for different contexts and situations [25]. The Software Radio technology [3] 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 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) [19]. 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 [10].

In the context of protocol stack reconfiguration, 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). However, by backward-compatible changes of a protocol component, the protocol reconfiguration can be carried out in the component independently without disrupting its peer component. As an example, consider a reconfiguration that changes TCP component in a running TCP/IP protocol stack into TCP-Vegas component [4]. TCP-Vegas is backward-compatible with TCP and therefore the reconfiguration can be performed in the stack independent of its peer stack.

There are some research activities for presenting dynamic protocol stacks. Some of them, such as [18, 29], provide reconfigurability at deployment-time (customizability) for protocol stacks. Some others including [16, 1] support reconfigurability at run-time for “idle” protocol stacks. In these works, safe states are points of protocols’ executions in which transactions of the corresponding protocol have been completed. However, in long-running servers, having long and important connections (e.g., TCP connections), it is unfavorable to wait until the end of protocol transactions. A few approaches, such as [13], support dynamic reconfiguration of “running” (not idle) protocol stacks. They can reconfigure running protocols in safe states that can be in the middle of protocol transactions. In these approaches, reconfiguration of a protocol component should be transparent in the peer component’s point of view.

In this paper, the reconfiguration problem is defined as changing one of the peer stacks at run-time transparently. Unlike the related work, we can reconfigure running protocol components in the middle of their protocol transaction. For such a reconfiguration, we propose a procedure for reconfiguration management and control. The procedure employs two ideas; we propose every protocol has a data structure for representing its state (called PCB); and protocol developers mark some states as safe states for starting possible reconfigurations.

The rest of this paper is organized as follows. Section 2 describes backgrounds about protocol execution and reconfiguration assurance in protocol stacks. In Section 3, we explain the proposed solution for backward-compatible reconfigurations in protocol stacks. Mechanisms for assurance and also the reconfiguration procedure are presented. In Section 4, we describe implementation and evaluation of the solution. 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 and explain assurance in reconfigurations of protocol stacks.

2.1 Protocol Execution

We consider a simple layered protocol stack model to describe the reconfiguration problem for communicating protocol components. Fig. 1 shows two communicating protocol stacks, (Stack1 and Stack2). In each layer of each stack, one independent entity which we refer to as protocol components (or in short components), provides functionality of its corresponding protocol. For instance, components P1 and P2 in the figure, provide functionality of the protocol P. We suppose components as an independent run-time entity. The protocol components interact with each other by exchanging protocol messages.

State of a protocol component describes all information related to 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 Reconfiguration Assurance

One of the important reasons for the lack of practical use of reconfigurable component-based systems is dealing with assurance of reconfigurations [28]. 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 [10]. Based on this notion, if the new component starts re-execution from a reachable state\(^1\), then the reconfiguration will be assured. Based on [10], 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. Requirements for such a safe state and state transfer are described below.

**Safe state** A safe state for a reconfiguration has been defined as a state having no interaction with the other components [11, 16]. A component in a safe state does not accept new requests, does not initiate new operations, and all its initiated operations have been completed [23].

There are two types of algorithms to find safe states in an execution. **Static algorithms**, such as [12, 33], use knowledge of the system structure to identify safe states. They always identify the same set of safe states for a particular reconfiguration. **Dynamic algorithms**, for example [9, 23, 5], use run-time knowledge such as components interactions to find safe states. Usually, dynamic algorithms disrupt small parts of a system than static algorithms.

**State transfer** Execution of the new component may be resumed from a non-initial state, which we refer to as the *restarting state*. Two approaches exist to transfer a state between two components, *direct state transfer* and *indirect state transfer* [31]. In the former approach, the new component uses the implementation of the old component to interpret and convert the state from the old component. In the latter approach, the old component exports its state in an abstract representation form which is used by the new component. For the state transfer, firstly, it is necessary to find the restarting state in the new component; secondly, the restarting state should be initialized to resume the execution.

In the following section, we explain the proposed solution for backward-compatible reconfigurations of running protocol components.

3 The Solution

In this section, we propose our solution for transparent reconfigurations of running protocol components. First, we describe reconfiguration supports for protocol components; then we explain required knowledge for such reconfigurations; finally, we state the proposed solution for the reconfiguration management and control.

3.1 Reconfiguration Support for Protocol Components

Reconfigurable protocol components should provide some extra functionalities to support dynamic reconfigurations. Every protocol component should implement the `ReconfigurableComponent` interface. Fig. 2 shows the required methods; `saveState()` and `restoreState()` methods are used for state transfer and `start()` and `stop()` methods are used to start and stop execution of the component. The `semiFreeze()` method should be implemented for each component in order to support freezing the component in a proper state.

Moreover, dynamic reconfigurations of protocol components require indirect communication between two

---

\(^1\)State \(s\) in component \(C\) is said reachable state, if and only if an execution of component \(C\) starting from an initial state can reach \(s\) at some time for some inputs.
adjacent protocol components (layers) in order to present transparent run-time reconfigurations [8]. 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 (Fig. 3).

3.2 Protocol Knowledge Base

As stated in the Subsection 2.2, an assured reconfiguration of a protocol component necessitates to freeze the component in a safe state and to initialize the new component completely. In order to achieve this goal, our idea is to keep enough information about reconfigurable protocols in a Protocol Knowledge Base (PKB) component. Specifically, PKB contains two types of knowledge, namely, role-compatible protocols, and protocol state information (Fig. 4), as explained in the following subsections.

3.2.1 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. Two protocols that can be reconfigured to each other are called role-compatible protocols. A pair of protocol and its backward-compatible version is a well-known example for role-compatible protocols.

For each pair of role-compatible protocols, PKB contains a state mapping function that maps corresponding macro-states of the two protocols. This helps reconfigurations as follows: when a protocol stops in a state (freeze state), through the state mapping function, a corresponding state (restarting state) can be found to resume the execution.

3.2.2 Protocol State Information

We present a model for state (micro-state) representation of protocol components by introducing protocol control block (PCB) that contains state information at run-time. 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 in DRAPS.

Moreover, for protocols in one category (a protocol and its extensions), we define basic PCB (BPCB) as an abstract PCB for the base protocol specified in standards and RFCs. For example, Fig. 5 shows the parameters for BPCB in original TCP protocol (RFC793). PKB contains a set of PCBs’ and BPCBs’ parameters and formats for different protocols.

PCB handlers are provided to set values of PCB parameters. In state transfer, the old PCB is used to evaluate the same parameters in the new PCB. Afterwards, the PCB handlers are used to complete the state transfer. PKB includes PCB handlers for different reconfigurations. Through PCBs and PCB-handlers we offer indirect state transfer.
3.2.3 TCP-SACK State Initialization

We give an example for building a PCB. TCP-SACK [15] is a selective acknowledgment extension for TCP\(^2\). It uses two TCP options, namely, SACK-PERMIT and SACK. These options are added to the TCP header. The first is an enabling option, which may be sent in a SYN packet to indicate the SACK option can be used once a connection is established. The second may be sent over an established connection, once permission has been given by SACK-permitted. The SACK option is sent by a data receiver to inform the data sender that non-contiguous blocks of data have been received and queued. The data receiver awaits the data reception in order to fill the gaps in the sequence space between received blocks. When missing parts are received, the data receiver acknowledges the data by advancing the left window edge in the Acknowledgement Number Field of the TCP header.

PCB for TCP-SACK can be built by adding two parameters, SACK-PERMIT and SACK, to the TCP BPCB. For state initialization in TCP-SACK, the TCP PCB is copied into the TCP-SACK PCB and the SACK-PERMIT parameter in the PCB is set through a PCB handler. Since the SACK parameter is calculated based on the received data, it is not necessary to be initialized.

3.3 Reconfiguration Management and Control

Reconfiguration management and control is provided through a component, called RMC. It can receive reconfiguration commands from different sources including system administrators, monitoring components in the system, or peer systems’ RMC. A reconfiguration command asks replacing an old protocol component with a new one. Accordingly, RMC freezes the old component in a safe state, installs a new component, and initializes it for re-execution from a restarting state.

3.3.1 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 component execution 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

\(^2\)Although TCP-SACK requires both endpoint protocol components to change in order to be of value, this example only explains state initialization.
input buffer are SRPs. In such points, either no operation has been started on the packet or all the operations have been completed. Usually, macro states of protocols are like this and we use them as SRPs. This definition is consistent with the definitions in [11, 34, 16], where all operations of the component in SRP should be completed. However, we do not restrict the component’s buffers to become free in SRPs. This implies great flexibility in finding SRPs, in comparison with the others such as [1]. In our solution, in order to find a SRP, it is not necessary to wait for the buffers to become empty.

Generally, finding a SRP in an execution of a component is a reachability problem, which is undecidable [10]. To cope with this problem, we ask protocol developers to “mark” some SRPs 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 normal execution mode, a component does not report SRPs. However, we introduce semi-freeze mode in which the component reports every SRP upon reaching (like in debug mode). 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.

3.3.2 The Reconfiguration Procedure

To simplify the problem of transparent reconfigurations of a running protocol component, we assume there is only one reconfiguration at a time and no concurrent reconfigurations can be initiated.

The flowchart of the reconfiguration procedure is depicted in Fig. 6. There are three preconditions to start a reconfiguration. First, the old component should have at least one SRP; second, mapping functions for the SRPs of the old component should have been defined in PKB; third, new PCB can be completely initialized through the old PCB and PCB handlers.

In the case of satisfaction of all preconditions, the RMC component starts installation and execution of a new component. It installs the new component and invokes the semiFreeze() method of the currently running component to start its semi-freeze mode. In this mode, RMC monitors the component execution and freezes it by invoking its stop() method in a determined SRP. 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 in order to transfer the protocol state. After that, the new component is re-executed through its start() method. From now, the new component is available for its user components and the application layer can send data through the new protocol.

It is worth noting that, when the old protocol component enters into the freeze mode, there maybe some packets inside the communication protocol stacks (packets that have been sent by the peer component but not delivered by the old component yet). These packets are buffered in upper/lower layers of the frozen component.

Figure 6. Flowchart for the reconfiguration procedure

4 Implementation and Evaluation

We have implemented a prototype of the RMC component\textsuperscript{3} to demonstrate feasibility of our solution in transparent reconfigurations of running protocol components. Overhead of using wrappers in protocol stack performance has been evaluated in related works such as [8, 24]. In this section, we describe the evaluation of the proposed mechanisms for assured and transparent dynamic reconfigurations. The evaluation includes the reconfiguration procedure and the overhead of finding safe states and using PCBs in state transfer.

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

\textsuperscript{3}An implemented prototype of the RMC is available at http://mehr.sharif.edu/~niamanesh/RG.htm.
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.

Figure 7. The experimentation environment

Two communicating peer protocol stacks are considered, Stack1 and Stack2 (Fig. 7). Stack1 is implemented using the proposed solution. Applications on the top of stacks exchange data with each other. Both applications use a light-weight version of TCP protocol, which have been implemented for the transport layer. The IP layer is simulated using two Linux FIFOs, one for outgoing data and the other for incoming data. In Stack1, wrappers for TCP and IP layers have also been implemented.

In the experiments, a reconfiguration command is sent to the RMC in Stack1 to initiate a transparent reconfiguration. The command is to replace running TCP with TCP-CA (TCP with Congestion Avoidance). TCP and TCP-CA are implemented such that all their macro-states are SRPs.

The TCP-CA PCB has two more parameters that are congestion window size (CWND) and slow start threshold (SSthresh). These parameters are evaluated using two PCB handlers to set their default values (CWND = 512 bytes, SSthresh = 65536 bytes).

Table 1. The performance of replacing TCP

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install</td>
<td>33</td>
</tr>
<tr>
<td>PKB Operations</td>
<td>3</td>
</tr>
<tr>
<td>Semi-Freeze</td>
<td>3</td>
</tr>
<tr>
<td>Freeze</td>
<td>17</td>
</tr>
<tr>
<td>Reconfiguration</td>
<td>56</td>
</tr>
</tbody>
</table>

The freeze time includes the time for state transfer. In the state transfer TCP-CA PCB is initialized based on TCP PCB and two PCB-handlers. Restricting the TCP “send buffer” to become empty for starting a reconfiguration, the state transfer takes up 17 milliseconds (5 milliseconds for copying the old PCB into the new one, and 12 milliseconds for two PCB-handlers). It is important to note that, the freeze time is the only period that both old and new protocol components are unavailable. The total reconfiguration time, which includes installation, PKB operations, semi-freeze, and freeze times, is 56 milliseconds.

We have not restricted buffers to become empty for starting a reconfiguration; therefore, the overall state transfer time depends on the amount of data in TCP “send buffer”. Based on our experiments state transfer of TCP with full buffers of the size 91KB and 133KB takes 43.5 and 71 milliseconds correspondingly.

Table 2 shows comparisons of our solution (RMC) and the two related works, which are implemented and evaluated in the similar environment. As shown in the table, the freeze period in our solution is much lower than the others. This is due to performing state transfer through PCB and PCB-handlers. In comparison, [13] uses Java serialization technique [17] which takes around 190 milliseconds. This shortening of the freeze time in our solution is achieved through the PCB model, which forces one-time development overhead for each protocol and presents a short time for the freeze period, which is critical to be short.

For the TCP protocol, since the timeout of TCP sockets is usually more than tens of seconds on different machines, the reconfiguration is performed trans-
It only presents customizibility of protocol frameworks. However, the provision of a library of classes, which can be used to build reconfigurable protocol stacks. DPF [1] 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.

In [13], 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 short freeze time.

## 6 Concluding Remark

This paper proposes a solution for dynamic reconfigurations of protocol stacks. Although dynamic reconfiguration for protocol stacks is not new, our work is novel in that it supports assured and transparent dynamic reconconfigurations for running protocol components. Unlike related work, we can reconfigure protocol stacks at deployment-time.

The DIPS/CuPS framework [16] is intended for the development of customizable system software. This research mainly concentrate on a component model for protocol components. Reflection points in DIPS/CuPS, which performs packet switching within a protocol stack, can only support run-time reconconfigurations. Safe states for reconconfigurations 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 reconconfigurations for protocol stacks with idle protocols. It does not provide flexible enough mechanisms for reconconfigurations of running protocols (reconfiguration during protocol transactions). DIPS/CuPS supports both direct and indirect state transfer mechanisms.

Dynamic Protocol Framework (DPF) [1] mainly concentrates on building an adaptive protocol stack by automatic discovery and selection of protocol components. It supports on-the-fly reconconfigurations of protocol stacks and provides a synchronization mechanism to ensure the compatibility of protocol stacks on communication peers. In DPF, safe states for reconconfigurations are restricted to be at the end of protocol transactions. Moreover, supporting mechanisms for state transfer are not clearly defined and addressed.

## 5 Related Work

Developing dynamic reconfigurable systems have been reported in the literature, such as CONIC [14], ARGUS [2], and POLYTH [11]; some others such as [7, 16] provide reconfigurable component models. In the context of reconfigurable protocol stacks, related work are mainly focused on implementing frameworks to support reconfigurable protocol stacks. Reconconfigurations in protocol components can be either at deployment-time, at run-time with idle protocols, or at run-time with running protocols.

Ioana Sora et al. propose the “protocol building block” description and protocol selection algorithms in [26]. They use an algorithm to select building blocks in case that all specified features are provided and all dependencies of selected components are satisfied. This paper offers protocol stack compositions at deployment-time.

In [18], 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.

### Table 2. The comparison of times of reconconfiguration steps in the related solutions (NA = Not Available)

<table>
<thead>
<tr>
<th>Solution/Time(ms)</th>
<th>Finding SRP</th>
<th>State Transfer</th>
<th>Recon.</th>
<th>Freeze Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPF [1]</td>
<td>NA</td>
<td>NA</td>
<td>200</td>
<td>≥ 200</td>
</tr>
<tr>
<td>[13]</td>
<td>3</td>
<td>190</td>
<td>214</td>
<td>≥ 190</td>
</tr>
<tr>
<td>RMC</td>
<td>3</td>
<td>5</td>
<td>56</td>
<td>5</td>
</tr>
</tbody>
</table>

As the timeout value for the most of communication protocols is much more than a second, we can expect that the presented solution can transparently reconconfigure 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.

In [18], 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 [16] is intended for the development of customizable system software. This research mainly concentrate on a component model for protocol components. Reflection points in DiPS/CuPS, which performs packet switching within a protocol stack, can only support run-time reconconfigurations. Safe states for reconconfigurations 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 reconconfigurations for protocol stacks with idle protocols. It does not provide flexible enough mechanisms for reconconfigurations of running protocols (reconfiguration during protocol transactions). DiPS/CuPS supports both direct and indirect state transfer mechanisms.

Dynamic Protocol Framework (DPF) [1] mainly concentrates on building an adaptive protocol stack by automatic discovery and selection of protocol components. It supports on-the-fly reconconfigurations of protocol stacks and provides a synchronization mechanism to ensure the compatibility of protocol stacks on communication peers. In DPF, safe states for reconconfigurations are restricted to be at the end of protocol transactions. Moreover, supporting mechanisms for state transfer are not clearly defined and addressed.

In [13], 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 short freeze time.
components during their transactions. Through modeling protocols’ states by PCBs, we have achieved a very short time for the freeze period. This presents enough transparency of reconfigurations in the peer component’s point of view. We believe, PCBs and BPCBs can be added to protocols standards and RFCs; therefore protocol developers can share PCBs and BPCBs easily.

For validation of the solution, we have implemented a prototype. Test scenarios for assured and transparent reconfigurations of the TCP protocol in a TCP/IP protocol stack are realized through the RMC component. The experimental results show maintaining of an acceptable transparency.

Future work can be carried out in several directions. Firstly, managing knowledge in PKB and specification of protocol components require proper tools. Secondly, automatic generating of SRPs in the source code of protocol components is feasible. Lastly, providing automatic detection and execution of transparent recon-figurations of the TCP protocol in a TCP/IP protocol stack are realized through the RMC component. The experimental results show maintaining of an acceptable transparency.

For validation of the solution, we have implemented a prototype. Test scenarios for assured and transparent reconfigurations of the TCP protocol in a TCP/IP protocol stack are realized through the RMC component. The experimental results show maintaining of an acceptable transparency.

Future work can be carried out in several directions. Firstly, managing knowledge in PKB and specification of protocol components require proper tools. Secondly, automatic generating of SRPs in the source code of protocol components is feasible. Lastly, providing automatic detection and execution of transparent configurations has a great importance in self-configuring of autonomic systems.

References


