

DRAPS: A Framework for Dynamic Reconfigurable Protocol Stacks

MAHDI NIAMANESH AND RASOOL JALILI

Department of Computer Engineering

Sharif University of Technology

Tehran, Iran

E-mail: {niamanesh@mehr.; jalili@}sharif.edu

Forthcoming networked systems require mechanisms for on-the-fly reconfiguration in their protocol stacks to be able to operate in different situations and networks. Since every protocol in a protocol stack has at least one peer protocol in another system, dynamic reconfiguration of a protocol raises the need for reconfiguration in the peer stack. For an assured dynamic (run-time) reconfiguration, executions of two peer protocols are stopped in a safe state, new protocols are initialized, and stacks switch to the new protocols at the same time. This paper proposes a software framework for dynamic reconfiguration of two communicating protocol stacks. A distributed algorithm is implemented in the framework of two communicating stacks in order to present an assured and synchronous reconfiguration of protocols. For demonstration, we have implemented a prototype of the framework and the algorithm to reconfigure two communicating TCP components by their secure version.

Keywords: protocol stack, protocol component, dynamic reconfiguration, single and distributed reconfiguration, safe reconfiguration point, state transfer

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 [16]. Software Radio technology [3] offers dynamic reconfigurability for protocol stacks of such systems and devices in order to facilitate applications such as changing network of a device due to its mobility, changing routing algorithms of switches, changing security modules in protocol stacks, bug fixing, and customizing the protocol stack of a device for better performance.

There are some research activities for presenting dynamic protocol stacks. Most of them, such as [7, 10], suppose a running protocol stack as a stand-alone system (not in a network). To provide an assured reconfiguration, they have defined the safe state of a running component, as a state where the component has no data (application data) and is not in any interaction with the other components [5, 10]. A few approaches have considered peer stacks during protocol stack reconfigurations. In the Software Radio domain, related work such as [11] have mainly concentrated on device reconfigurations through a server (*e.g.*, changing the air interface of a terminal through a network).

In this paper, the reconfiguration problem is defined as *changing of two peer stacks at run-time synchronously*. Unlike the related work, we reconfigure two peer stacks while they are communicating. Our idea is to keep enough information about protocols to per-

Received July 31, 2007; revised February 29, 2008; accepted April 17, 2008.

Communicated by Makoto Takizawa.

form synchronous and assured reconfigurations. For such a reconfiguration, we propose a software framework including an algorithm for reconfiguration management and control. Through the proposed framework each stack can request a synchronous reconfiguration with its peer while they are communicating. The framework guarantees smooth change of the old protocol of the stack to the new one.

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 framework for dynamic-reconfigurable architecture for protocol stack (DRAPS). The main design issues of DRAPS including mechanisms for assurance and also the reconfiguration algorithm are presented in section 4. In section 5, we describe implementation and evaluation of the framework. In section 6, we discuss related work and section 7 concludes the paper.

2. BACKGROUND

2.1 Protocol Execution

We consider a simple model for describing the reconfiguration problem in communicating protocol stacks. In each layer of the stacks, one component, which we refer to as *protocol component*, implements functionality of its corresponding protocol. 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.

Execution of a protocol component is started from an *initial state*. Sending or receiving protocol messages changes the execution state of the protocol component. Execution proceeds in the states until reaching a *final state*, which indicates the completion of the execution. During the execution, two peer components exchange protocol messages and data in some *compatible* states. Two states from two components are compatible if all inputs and outputs of the components on those states “match” each other. We have formally specified two compatible states in [13].

For simplicity, we split the state of a protocol component into macro-state and micro-state. Macro-state describes the states of the protocol’s finite state machine. Examples of macro-states in the TCP protocol are CLOSED and ESTABLISHED. Micro-state describes the state of the protocol at run-time, to maintain information for operations like reliability, error handling, and congestion control. We use macro-states of components in the specification level and micro-states at the execution level.

2.2 Reconfigurations in Protocol Stacks

In two communicating protocol stacks, dynamic reconfiguration in a component may lead to a corresponding reconfiguration in its peer component. In this point of view, we can categorize protocol stack reconfigurations into two types; *single reconfiguration* and *distributed reconfiguration*. Fig. 1 depicts a scenario for single reconfigurations and a scenario for distributed reconfigurations. As depicted, in single reconfiguration (part

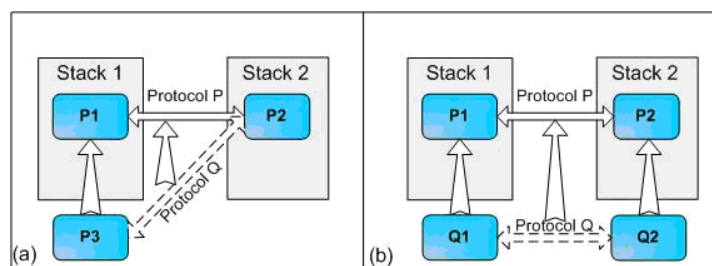


Fig.1. Types of dynamic reconfigurations for protocol stacks.

(a) of the figure) only one of the peers attends in the reconfiguration and in distributed reconfiguration (part (b) of the figure) both peers attend in the reconfiguration.

Fig. 1 (a) depicts a situation that 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, P_1 , is replaced by a new component P_3 that can safely interoperate with the old peer component, P_2 , in the peer stack. In this case, the new component, P_3 , is backward-compatible with the old one. Therefore, the reconfiguration can be carried out in the changing stack without interrupting the peer stack.

Fig. 1 (b) depicts a scenario that two peer components in two stacks synchronously change into another protocol. For two communicating TCP/IP protocol stacks, changing TCP protocol of two stacks into SCTP protocol [18] can be an example of this type.

2.3 Reconfiguration Assurance

One of the important reasons for the lack of practical use of reconfigurable component-based systems is dealing with assurance of reconfiguration [19]. 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. This notion is stated by Gupta in [4]. Based on this notion, if the new component starts re-execution from a *reachable state*¹, then the reconfiguration would be assured. Based on [4], to have a reachable state, the running component should be frozen in a safe state, and the new component should resume the execution from an initialized state.

Considering two communicating components as a distributed component, its state is the *global state* of the two components. Based on Gupta's notion for the assurance, the reconfiguration of two communicating components is assured if after the reconfiguration, the execution resumes from a reachable global state. We explain requirements for such a global state in following:

Safe state A safe state for a reconfiguration has been defined as a state that has no interaction with the other components [5, 10]. 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.

¹ State s in component C is said reachable, if and only if an execution of component C starting from an initial state can reach s at some time for some inputs.

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

3. DRAPS OVERVIEW

DRAPS (Dynamic-Reconfigurable Architecture for Protocol Stack) is an extendable framework² that presents assured and synchronous dynamic reconfiguration for two communicating protocol stacks³.

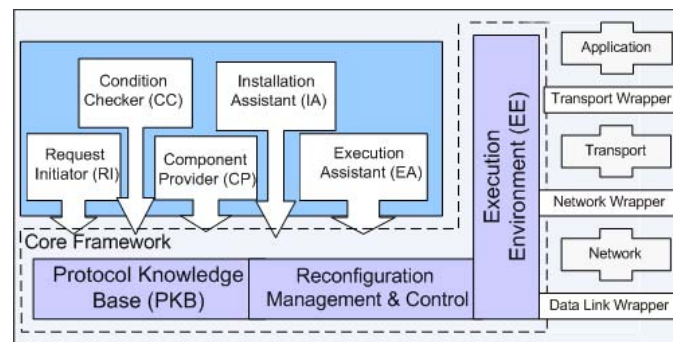


Fig. 2. Architecture of DRAPS including core framework and plug-ins.

Architectural components of DRAPS are shown in Fig. 2. As shown, DRAPS is built out from a core framework and some plug-in components. The core framework is responsible to perform assured and synchronous dynamic reconfigurations and consists of three components, namely, Reconfiguration Management and Control (RMC), Protocol Knowledge Base (PKB), and Execution Environment (EE). RMC provides mechanisms for assurance and synchronization of two peer stacks. PKB is a knowledge base component, responsible for initializing protocols specifically when they are resumed from non-initial states. EE is a component for installation and execution of components. Plug-in components in DRAPS present extendability for the framework by providing supplementary requirements of dynamic reconfigurations. DRAPS frameworks in two peer stacks cooperate with each other in order to reconfigure two peer components synchronously. A distributed reconfiguration algorithm is executed by RMCs in the frameworks. The algorithm has some messages which are exchanged between two RMCs.

² An implemented prototype of the framework is available at <http://mehr.sharif.edu/~niamanesh/RG.htm>.

³ The initial versions of DRAPS have been published in [12] and [14].

4. DRAPS DESIGN

4.1 Reconfiguration Support for Protocol Components

Protocol components in DRAPS should provide some extra functionalities to support dynamic reconfiguration. For this purpose, DRAPS presents an implementation-level component model for reconfigurable protocols. Every protocol component should implement the ReconfigurableComponent interface. As shown in Fig. 3, saveState() and restoreState() methods are used for state transfer. The start() and stop() methods are used to start and stop execution of the component. The semiFreeze() method helps the component to start reporting its state to the framework.

```
Public interface ReconfigurableComponent {  
    void saveState(String path);  
    void restoreState(String path);  
    void start();  
    void stop();  
    void semiFreeze();  
}
```

Fig. 3. DRAPS component model.

4.2 Protocol Wrappers

A wrapper is a component that implements the interface of a service. It is used to provide an indirect communication between two components in order to present a transparent run-time reconfiguration. Every reconfigurable component in DRAPS should have one wrapper to handle requests from the component. For example, a TCP wrapper that implements TCP interface, can manage the application's requests from the TCP component.

4.3 DRAPS Plug-in Components

Based on the common supplementary requirements for a dynamic reconfigurable software system, we have identified some types of plug-in components for DRAPS. *Request Initiator* is a component type to detect the need for reconfiguration initiation. *Component Provider* is another type of DRAPS plug-in, which provides required components in a dynamic reconfiguration. Other types of plug-in components can be defined either.

4.4 Protocol Knowledge Base

As stated in section 2.3, assured reconfiguration for two communicating components necessitates to freeze the components in a safe global state and to initialize new components completely. Our idea to achieve this goal is to keep enough information about reconfigurable protocols in the framework. The Protocol Knowledge Base (PKB) component provides all required knowledge for this purpose. It initializes the restarting state (the non-initial state in which the component should resume the execution) in the

new component and finds a safe global state for freezing. Specifically, it contains three types of knowledge, namely, peer-compatible protocols, role-compatible protocols, and protocol state information. In following, we explain in detail.

4.4.1 Peer-compatible protocols

Synchronous reconfiguration requires commitment of two peer stacks on both change and freeze state. Commitment on the change causes two peers change into compatible protocols. For example, when a peer changes into TCP protocol, the other peer can not choose UDP protocol for its transport layer. Accordingly, PKB contains all pairs of compatible protocols (peer-compatible protocols). Commitment on the freeze state necessitates two peers to start the reconfiguration in a safe global state. For this purpose, the PKB component keeps all compatible macro-states for each pair of peer-compatible protocols. 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.

4.4.2 Role-compatible protocols

In a reconfiguration of an existing component into a new component, the new one should play the same role as the old 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.

For each pair of role-compatible protocols, PKB contains a state mapping function that maps corresponding macro-states of the two protocols. This helps RMC in reconfiguration as follows; when a protocol stops in a state (freeze state), through the state mapping functions RMC finds a corresponding state (restarting state), to resume the execution. For this reason, the two protocols should be role-compatible and their mapping functions should be defined in PKB.

4.4.3 Protocol state information

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 programmers 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. 4 shows the parameters for BPCB in original TCP protocol (RFC793). PKB contains a set of PCBs' and BPCBs' parameters and formats for different protocols.

DRAPS provides PCB handlers to set values of PCB parameters. They are also used to set values of remote PCB 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.

Variables(V)

- A. Addressing Variables
 - SRC: source port
 - DST: destination port
- B. Segment Variables
 - SEG.SEQ: segment sequence number
 - SEG.ACK: segment acknowledgement number
 - SEG.LEN: segment length
 - SEG.WND: segment window (Receiver Advertised Window)
 - SEG.CTL: control bits (ACK, RST, SYN, FIN)
- C. Send Sequence Variables
 - SND.UNA: send unacknowledged
 - SND.NXT: send next
 - SND.WND: receive window
 - ISS: initial send sequence number
- D. Receive Sequence Variables
 - RCV.NXT: receive next
 - RCV.WND: receive window
 - IRS: initial receive sequence number
- E. Timers
 - REXMT: Retransmission Timer
 - TIMWAIT: Time-wait Timer
 - USERTIME: User Timer
- F. Counters
 - dACK: duplicate ACK counter
 - ExpBoff: exponential backoff counter
- G. Other
 - CurrState: Current State
 - PreState: Previous State
 - RTO: Retransmission Timer Out value
 - RTT: Round Trip Time, used to calculate RTO
 - SRTT: Smoothed RTT, used to calculate RTO
 - CWND: Congestion window
 - MSS: Maximum Segment Size
 - SStresh: Slow Start Threshold
 - MSL: Maximum Segment Lifetime
- H. Buffers
 - SND.Buff: Send Buffer
 - RCV.Buff: Receive Buffer
 - OO.RCV.Buff: Out of Order Receive Buffer
 - SND.UNAQ: Holds sent but unacknowledged segments
 - UCallQ: Holds outstanding user calls

(e.g., SEND, RECEIVE, CLOSE)

Fig. 4. TCP parameters in a BPCB.

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 transferring and finally the remote PCB handlers from the peer are applied.

4.4.4 TCP-SACK state initialization

We give an example for building a PCB. Consider TCP-SACK [9], selective acknowledgment extension for TCP, which uses two TCP options, namely, SACK-PERMIT and SACK. These options are added to the header part of TCP. The first is an enabling option, which may be sent in a SYN segment to indicate that 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.

PCB for TCP-SACK can be built by adding two parameters, SACK-PERMIT and SACK, to 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 SACK parameter is calculated based on the received data, it is not necessary to be initialized.

4.5 Reconfiguration Management and Control

Reconfiguration Management and Control (RMC) component is responsible for reconfiguration management and control in DRAPS. It can receive reconfiguration command from different sources including system administrators, monitoring components in the system, or peer system RMCs.

The reconfiguration process is started upon receiving a reconfiguration command, which comprises of all necessary information to perform the reconfiguration. This information includes a map that describes the change, a state of running component for starting the reconfiguration, and some PCB handlers to evaluate the restarting state in the new component. The reconfiguration map includes a set of reconfiguration operations for changing, such as adding or removing the components. It is provided based on the peer-compatible protocols in the PKB or a reconfiguration administrator. The state for the reconfiguration can be either provided by the system administrator or can be determined through the PKB component.

For a synchronous reconfiguration of two communicating components, RMCs in two stacks interact with each other. They freeze the intended running components in a safe global state, install new components and initialize them for re-execution from the restarting states. RMC exploits the EE and PKB components for installation and initialization of new components, respectively.

4.5.1 Reconfiguration channel

An important point in dealing with dynamic reconfiguration in protocol stacks is that, the reconfiguration causes communication inside the stack become limited or even blocked. For synchronous reconfiguration of two peer stacks, this can lead to the problem in communication of their RMCs during a reconfiguration. To cope with this problem, some related work has been proposed a dedicated reconfiguration channel for dynamic reconfigurations [6]. In these approaches, every reconfigurable networked system should have a dedicated *reconfiguration channel* (protocol stack), besides a usual protocol stack. The reconfiguration channel is used to download new components and to communicate with the system during a reconfiguration.

In contrast, RMC component in DRAPS uses an existing protocol stack of the system for its communications with the peer RMC during a reconfiguration. Although this communication channel is blocked during the freeze period (due to a change in a protocol component in the stack), we provide such a distributed reconfiguration algorithm not requiring communication of peer RMCs in the blocked period. In this way, we do not need a dedicated channel for reconfigurations. This removes the cost of a dedicated reconfiguration channel for reconfiguration management and control.

4.5.2 Finding a safe global state

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 points, either no operation is started on the packet or all the operations are completed. This definition is consistent with the definitions in [5, 10], where all operations of the component in SRP should be completed. However, we do not restrict the component's buffers to become free in SRP.

Generally, finding a SRP in an execution of a component is a reachability problem, which is undecidable as stated in [4]. To cope with this problem, we ask protocol developers to “mark” some SRPs in the source code of protocols. A good set of SRPs for a protocol is the set of its macro-states. 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.

However, not every SRP in a component is a safe state in point of the peer component view. A state in a component is safe for the peer component if it is compatible with the peer's current state. In other words, a global state (s, r) for two communicating components is a *safe global state* if s and r are locally SRP and their mapped states, denoted as s' and r' respectively, are compatible with each other.

In order to freeze a component in a SRP, we introduce *semi-freeze* mode for executions of protocol components. An execution in semi-freeze mode is the same as an execution in normal mode except that in this mode the component also reports its state in all determined SRPs.

To find a safe global state for a reconfiguration, for each reported state, say s , in semi-freeze mode, RMC checks whether its mapped state, *e.g.*, s' , has a compatible state in the peer component. If so, the pair of reported state and its compatible state in the peer component is the safe global state for the reconfiguration.

4.5.3 The reconfiguration algorithm

To simplify the problem of synchronous reconfiguration of two communicating peer components, we consider two assumptions:

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

In order to explain the proposed distributed algorithm for reconfiguration of communicating peer components, suppose two communicating protocol stacks exchanging application data. The algorithm has three types of messages (START, RECON, and EoREC) for communication between two RMCs in peer stacks. The START message is a protocol specific command for reconfiguration initiation and the RECON (RECONfigure) message is used by that initiator RMC to indicate the committed reconfiguration. Each RMC sends the EoREC (End of REConfiguration) message to the peer RMC after finishing the reconfiguration.

The sequence diagram of the algorithm is depicted in Fig. 5. The algorithm is started upon receiving a START message by a RMC. The START message includes a set of all possible reconfigurations for the receiver stack, each of which is a triple, (m_i, s_i, δ_i) , consisting of a map, a freeze state, and a list of PCB values, respectively. RMC prepares a corresponding START message containing all possible reconfigurations of the peer and sends to it. Upon receiving START by the peer RMC, it starts this algorithm as well. After sending the START message to the peer, both RMCs wait for a response message.

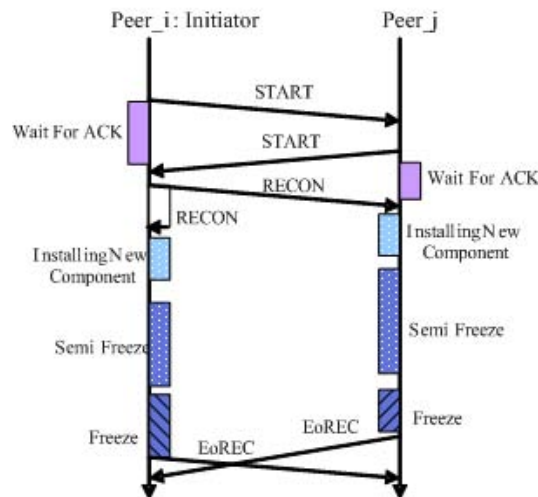


Fig. 5. Sequence diagram for the dynamic reconfiguration.

The initiator RMC receives a START message as the response for the sent START. It prepares RECON messages including the committed reconfigurations and sends them to the peer and itself, correspondingly. The peer RMC receives a RECON message as the response. Upon receiving the RECON message, each RMC invokes `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. Then, 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. Through the new stack, each RMC sends a EoREC message to its peer to indicate the reconfiguration completion. If

the peer component is not ready to receive packets, they will be kept in the output buffer of lower layer protocol in the peer stack. According to the algorithm, in the freeze period, RMC components do not require any communication with each other.

The START message for a reconfiguration synchronizes two peers in terms of the change map, state initialization, and freeze state. The initiator stack offers all possible reconfigurations for the peer through the START message and the peer sends back a set of all acceptable reconfigurations of the initiator based on the peer-compatibility check. Returning a START message with an empty reconfiguration map indicates that the peer refuses the proposed reconfiguration. The initiator stack selects one of the acceptable reconfigurations for itself and sends its compatible reconfiguration for the peer through the RECON message.

5. IMPLEMENTATION AND EVALUATION

In this section, we evaluate the proposed mechanisms for assurance and synchronization in dynamic reconfiguration. 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 a Centrino 1.5 GHz IBM personal computer with 256 MB memory. Linux (Debian 3.1 distribution) is used as the operating system.

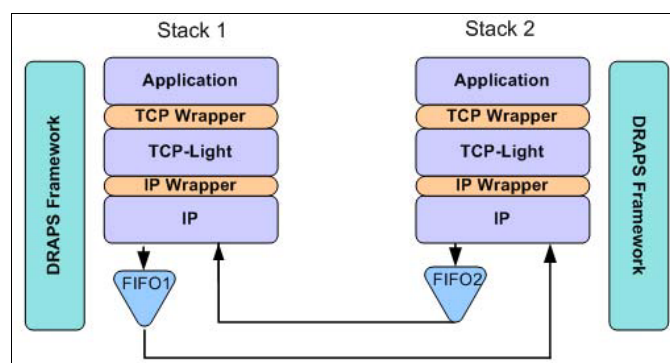


Fig. 6. The experimentation environment.

We consider two peer stacks, $Stack_1$ and $Stack_2$, as two communicating protocol stacks (Fig. 6). Applications on the top of each stack exchange data with each other. Both applications use a light-version of TCP protocol, that we have implemented based on DRAPS component model, for the transport layer. Wrappers for TCP and IP layers have also been implemented. The IP layer is simulated using two Linux FIFOs, one for outgoing data and the other for incoming data.

At first, the application on $Stack_1$ opens a connection via TCP and sends a file. $Stack_2$ on the other side receives the packets and writes them down in a file. The overall transfer time is calculated as t_1 . Moreover, the transfer time is calculated in presence of a reconfiguration during the transfer. After 60 percent of the file transfer, $Stack_1$ receives a

reconfiguration command to replace the TCP component with a SecureTCP to be able to encode the sent packets. We suppose SecureTCP is the same as TCP except that it also has two different options for coding packets. Both TCP and SecureTCP components are implemented such that two SRPs, namely, ESTABLISHED and CLOSED are embedded in their source codes. The command asks RMC to reconfigure the running TCP component in the ESTABLISHED state into the SecureTCP component. RMC component in $Stack_1$ composes a reconfiguration command including a map, replacement of TCP with SecureTCP, a compatible state in the peer, ESTABLISHED, and initialization parameters, a cryptography coding. The command is sent as a START message to the peer through the $Stack_1$. The peer stack replies via a START message and confirms the freeze state, map, and cryptography coding.

After the reconfiguration, the file transfer is resumed and the overall transfer time is calculated as t_2 . Reconfiguration time includes the time for installation, semi-freeze, and freeze. It is important to note that, the freeze period Freeze in Fig. 5 is the only time that the communication is blocked. Since we have not restricted the buffers to become empty for starting a reconfiguration, the overall state transfer time is depended on the amount of data in buffers. This condition implies great flexibility in finding SRPs, in comparison with others such as [1]. In order to find a SRP, in DRAPS, it is not necessary to wait for the buffers to become empty.

Table 1. The performance of replacing TCP.

	Time (ms) 133	Time (ms) 91
Install	33	33
PKB Operations	6	6
Semi-Freeze	214	168.5
Freeze	71	43.5
Reconfiguration	370	307
Normal Transfer (t_1)	2189.25	1521
Rec. in Transfer (t_2)	2281.25	1617.75
Rec. Overhead	0.04	0.05

Table 1 shows the performance of TCP reconfiguration according to the steps in our reconfiguration algorithm. Our experiments are based on two different buffer sizes in TCP; in the second column of the table, numbers are for the size of 133Kb and the third column are for 91Kb of buffer size. All numbers are averaged over a large number of iterations. Overhead of the dynamic reconfiguration can be calculated by $(1 - t_2/t_1)$ ratio, which is less than 5 percentage in our experiments. The default value for “send buffer” in TCP standard is 8Kb. By restricting the TCP “send buffer” to become empty for starting a reconfiguration, our experiments show that the average amount of freeze period is very low and less than 5 milliseconds. The amount of freeze period in our experiment is comparable with Java serialization and deserialization techniques for state transfer, which takes 179 milliseconds as stated in [7]. In the table, “PKB operations” is the time to compose appropriate START message through the PKB component. In our experiments, it takes up around 6 milliseconds. This time depends on the PKB structure.

6. RELATED WORK

Developing dynamic reconfigurable systems have been reported in literature, such as CONIC [8], ARGUS [2], and POLYLITH [5]. In the context of reconfigurable protocol stacks, related work are mainly focus on implementing frameworks to support reconfigurable protocol stacks. Ioana Sora *et al.* propose the “protocol building block” description and protocol selection algorithms in [17]. They use an algorithm to select building blocks in case that all specified features are provided.

The DiPS/CuPS framework [10] aims at the development of customizable system software. Authors mainly concentrate on a component model for protocol components. Reflection points of this framework, which performs packet switching within a protocol stack, can only support run-time reconfiguration. Safe states for reconfigurations are defined based on protocol transaction; that is, when the protocol execution is completed and buffers are empty the state is safe for a reconfiguration.

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 reconfiguration of the protocol stack and provides the synchronization mechanism to ensure the compatibility of protocol stacks on communication peers. In DPF safe states for reconfiguration and supporting mechanisms are not clearly defined. There is no mechanism for state transfer between the old protocol and the new one.

In [7], authors have proposed a framework that allows the programmer to create, to remove, and to replace protocol modules at run-time. The framework aims to preserve the module state as a data structure that manages the existing connections. To achieve these goals, a Java-based component framework is developed so that the programmers are able to implement their components under the proposed framework. This framework can dynamically reconfigure the components in a safe state and can help the components transfer their states. This framework does not support peer synchronization.

In the context of synchronous reconfiguration of communicating peers, in [15], authors propose an algorithm and a model for synchronous reconfiguration 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, the global update is performed when the protocol execution has been completed. Therefore, there is no need for finding global states and state transfer.

7. CONCLUDING REMARK

This paper proposed a framework for dynamic reconfigurations of protocol stack. Although dynamic reconfiguration is not new, our work is novel in that DRAPS is the first protocol stack framework that supports assured and synchronous dynamic reconfiguration for two communicating stacks. For such a reconfiguration, a distributed algorithm is implemented through two RMC components. The algorithm does not require any communication between two RMCs during the freeze period, in which the stack is blocked. It can change its underlying protocol stack layers without any problem in communication with the peer RMC. For validation, we have implemented a prototype of DRAPS framework. Test scenarios for assured and synchronous reconfigurations of TCP protocol in TCP/IP protocol stack are realized through the framework.

ACKNOWLEDGMENT

We would like to thank Serwah Sabetghadam, Reza Yousefzade Rahaghi, and Kame-
lia Asadzade Manjili for their comments on an earlier version of the paper.

REFERENCES

1. L. An, H. K. Pung, and L. Zhou, "Design and implementation of a dynamic protocol framework," *Journal of Computer Communications*, Vol. 29, 2006, pp. 1309-1315.
2. T. Bloom and M. Day, "Reconfiguration and module replacement in Argus: Theory and practice," *IEE Software Engineering Journal*, Vol. 2, 1993, pp. 102-108.
3. V. G. Bose, A. B. Shah, and M. Ismert, "Software radios for wireless networking," in *Proceedings of INFOCOM*, Vol. 3, 1998, pp. 1030-1036.
4. D. Gupta, "On-line software version change," Ph.D. Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, 1994.
5. C. Hofmeister and J. M. Purtilo, "Dynamic reconfiguration in distributed systems: Adapting software modules for replacement," in *Proceedings of International Conference on Distributed Computing Systems*, Vol. 7, 1993, pp. 101-110.
6. M. Laddomada, "Reconfiguration issues of future mobile software radio platforms," *Wireless Communications and Mobile Computing*, Vol. 2, 2002, pp. 815-826.
7. Y. Lee and R. Chang, "Developing dynamic-reconfigurable communication protocol stacks using Java," *Software Practice Experience*, Vol. 6, 2005, pp. 601-620.
8. J. Magee, J. Kramer, and M. Sloman, "Constructing distributed systems in conic," *IEEE Transactions on Software Engineering*, Vol. 6, 1989, pp. 663-675.
9. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," RFC 2018, 1996.
10. S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten, "DiPS: A unifying approach for developing system software," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001, pp. 175.
11. K. Moessner, S. Hope, P. Cook, W. Tuttlebee, and R. Tafazolli, "The RMA – A framework for reconfiguration of SDR equipment," *IEICE Transactions on Communication*, Vol. E85-B, 2002, pp. 2573-2580.
12. M. Niamanesh and R. Jalili, "A dynamic-reconfigurable architecture for protocol stacks of networked systems," in *Proceedings of the 31st Annual IEEE International Computer Software and Applications Conference*, Vol. 1, 2007, pp. 609-612.
13. M. Niamanesh and R. Jalili, "Formalizing compatibility and substitutability in communication protocols using I/O-constraint automata," in *Proceedings of IPM International Symposium on Fundamentals of Software Engineering*, LNCS 4767, 2007, pp. 49-64.
14. M. Niamanesh, S. Sabetghadam, R. Y. Rahaghi, and R. Jalili, "Design and implementation of a dynamicreconfigurable architecture for protocol stack," in *Proceedings of IPM International Workshop on Foundations of Software Engineering (Theory and Practice)*, LNCS 4767, 2007, pp. 396-403.
15. O. Rutti, P. T. Wojciechowski, and A. Schiper, "Structural and algorithmic issues of dynamic protocol update," in *Proceedings of the 20th IEEE International Parallel*

and *Distributed Processing Symposium*, 2006, <http://infoscience.epfl.ch/record/63676/files/>.

16. M. Satyanarayanan, "Pervasive computing: Vision and challenges," *IEEE Personal Communications*, Vol. 8, 2001, pp. 10-17.
17. I. Sora, S. Michiels, and F. Matthijs, "Policies for dynamic stack composition," Technical Report No. CW 313, Department Computer Science, Katholieke Universiteit Leuven, 2000.
18. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, "Stream control transmission protocol," RFC 2960, 2000.
19. C. Szyperski, "Component technology: what, where, and how?" in *Proceedings of the International Conference on Software Engineering*, 2003, pp. 684-693.



Mahdi Niamanesh received a B.S. in Computer Engineering in 1999. He continued his studies under the tutelage of Rasool Jalili, and received a M.S. in 2001. Now, he is a Ph.D. student in the Computer Engineering program at Sharif University of Technology with interests in dynamic reconfigurable protocol stacks and pervasive computing. While a doctoral student, he was teacher assistant for the Distributed Systems course in the university for four years.



Rasool Jalili was born in Iran in 1961. He received his bachelor's degree in Computer Science from Ferdowsi University of Mashhad in 1985, and his master's degree in Computer Engineering from Sharif University of Technology in 1989. He obtained his Ph.D. in Computer Science from the University of Sydney, Australia, in 1995. He then joined the Department of Computer Engineering at Sharif University of Technology, Tehran, Iran, in 1995. He has published more than 130 papers in Computer Security and Pervasive Computing in international journals and conference proceedings. He is now an Associate Professor. His research interest includes such areas as access control, vulnerability analysis, and database security, which he conducts at his network security laboratory (NSC, nsc.sharif.edu). He is the ISeCure journal (www.isecure-journal.org) Editor-in-Chief.