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 staring possible reconfigurations.

The rest of this paper is organized as follows. Section 2 describes backgrounds about protocol executions 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, ($Stack_1$ and $Stack_2$). 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 P_1 and P_2 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

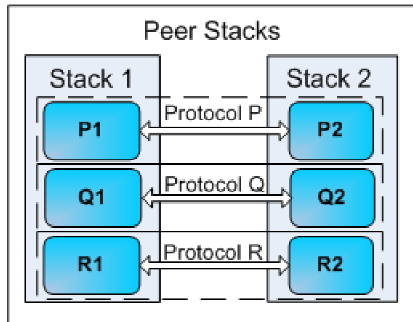

Figure 1. An example of two communicating protocol stacks

[6]. 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.

[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.

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