DR-TCP: Downloadable and reconfigurable TCP

Jae-Hyun Hwang a,*, Jin-Hee Choi b, Se-Won Kim a, Chuck Yoo a

a Department of Computer Science and Engineering, Korea University, Anam Science building 236, Anam-dong 5th Street, Seoul, South Korea
b Telecommunication R&D center, Telecommunication Network Business, Samsung Electronics, Dong Suwon P.O. Box 105, 416, Maetan-3dong, Suwon, Gyeonggi-do, South Korea

Received 13 May 2006; received in revised form 18 April 2007; accepted 7 May 2007
Available online 17 May 2007

Abstract

Advances in communication technology allow a variety of new network environments and services available very rapidly. Appearance of various network environments tends to enable a user with a mobile terminal to access among different networks simultaneously. However, since new network environment affects performance of communication protocols, terminal systems should provide adaptation schemes for the protocols in order to keep the quality of network performance high. A possible solution is to make the protocol reconfigurable to be adapted to current network environment. Unfortunately, because most existing network systems are implemented monolithically, they cannot support protocol reconfiguration dynamically at runtime.

This paper proposes a new reconfigurable model that enables TCP functions to be adapted whenever network environment is changed. The proposed scheme also supports binary-level protocol upgrade for extensibility by downloading new TCP variants which the terminal does not have for new network environment, and it is more suitable for mobile hand-held devices than existing source-level solution. Our model is based on a recursive state machine. We re-implement TCP Reno from scratch using our proposed model. The new implementation of TCP Reno is named DR-TCP. To demonstrate the effectiveness of DR-TCP, dynamic reconfiguration is performed over Internet, which successfully converts DR-TCP to TCP Westwood at runtime.

© 2007 Elsevier Inc. All rights reserved.

1. Introduction

Recent advances in communication technology lead to appearance of several types of networks, and as a result compound heterogeneous networks are getting popular. These networks consist of high-speed wired networks, wireless networks which include WLAN, cellular, satellite, Bluetooth, ad-hoc networks, and so on. The wireless environment enables a terminal to move, and such mobility provides the user to access among different networks easily at need.

Under the circumstances, network protocols in the terminal should consider characteristics of all kind of networks to keep the quality of network performance high. Most protocols have suffered performance degrading whenever they meet new network environment, and subsequently enhanced algorithms adapted to the environment have been suggested (Bakre and Badrinath, 1995; Durst et al., 1996; Floyd et al., 2000; Ludwig and Katz, 2000; Ratnam and Matta, 1998; Wang et al., 2002). This means that when the network environment changes, the network protocol also should be adapted properly. For instance of TCP, when the network environment is converted from wired to wireless, the protocol suffers from severe performance drop. In such case, definitely it is desirable that TCP variants for wireless network are applied. A problem is that there are so many types of wireless networks, and each has specific adaptation methods. It is impossible that single adaptation solves protocol adaptation problem in compound heterogeneous networks.
However, the existing network systems do not support protocol adaptation while users can access various networks through the mobility. This is because most network systems are monolithic implementation impossible to support reconfiguration. As a result, though adaptation solutions for each network environment already have been proposed and implemented, they are not practically used in real world. Addressing such limit, we approach for this problem through reconfiguring the protocol dynamically. Note that we focus on TCP reconfiguration in this paper since TCP is one of the most important internet protocols.

To address such TCP reconfiguration under mobility, we re-design TCP implementation. Considering that mobile devices generally have limited processing power and resource, our implementation model focuses on partial reconfiguration of TCP. With low reconfiguration overhead, this approach can improve TCP performance by reconfiguring only specific part of adapted TCP variants, not whole code.

In addition, we consider protocol extensibility by downloading new TCP variants which the terminal does not have for new network environment. TCP update solutions using mobile code downloading are already proposed, but they only support source-level update requiring code compilation (Patel et al., 2003a,b). These methods would not be practical to hand-held devices because they usually do not have development environment for compiling the protocol source code. The proposed method in this paper supports binary-level protocol downloading update, so it can reconfigure new protocol code immediately after downloading and loading the code into memory not requiring compiling the code, restarting the application or rebooting the system. This approach can also reduce downloading cost because the downloading code is much smaller than the case of source-level solution.

In summary, to the best of our knowledge, there is no model to enable a protocol to be partially reconfigured at runtime. The purpose of this paper is to propose a new model and show how TCP can be partially reconfigured based on the model. A main usefulness of DR-TCP is its dynamic nature in which TCP functionalities can be replaced and/or extended without stopping TCP operation. In other words, DR-TCP is operational while reconfiguration (or an upgrade) is in progress, which can make the kernel non-stopping. Another usefulness of DR-TCP is for devices where a compiling environment (for upgrade) is not expected. One example is mobile phone, and compiling is not something that a mobile phone is supposed to do.

The proposed model is based on recursive state machine and object delegation. State machine has been used in modeling a protocol theoretically but we use it to model functions of a protocol code. In our implementation model, we delegate the state machine's event handling to another state machine and consequently let the protocol functions separated easily by states. By making a protocol function an object, we can easily reconfigure a function of protocol by replacing the object.

The rest of the paper is organized as follows. Section 2 presents related work of network adaptation. In Section 3, we explain the concepts of recursive state machine model and object delegation. Based on the concept, Section 4 describes the architecture of our framework and downloadable protocol component. Section 5 shows how TCP can be restructured based on our model. And then we actually implement TCP named DR-TCP and address some implementation issues in Section 6. Section 7 presents reconfiguration loader. Section 8 performs experiments and analyzes the results. Finally we conclude the paper in Section 9.

2. Related work

Related work includes research on configurable network systems and transport protocols.

The x-kernel (Hutchinson and Peterson, 1991) is a well-known architecture for supporting various network protocol configurations. In the x-kernel, each protocol is wrapped up in Uniform Protocol Interface, and a protocol graph which defines usable protocols at each layer is configured at build time. This mechanism makes the network stack configurable easily, but it is static and coarse-grained configuration.

While the x-kernel configures the network stack at the granularity of the entire protocol, TIP (Boecking, 1993), F-CSS (Zitterbart et al., 1993), and ADAPTIVE (Schmidt et al., 1993) configure the network protocol by fine-grained modules. TIP divides the whole network service into fine-grained configurable modules, and combines them into one specific protocol based on application requirements. So TIP can provide different services for applications. F-CSS and ADAPTIVE have similar configuration methods. However, because the application requirements are rarely changed, those systems do not consider dynamic reconfiguration. That is, if a configured protocol needs to reconfigure its function partially in runtime, the system should restart the protocol configuration process in order to include an alternative module. Finally, Cactus (Hiltunen et al., 1999) and CTP (Wong et al., 2001) configure a transport protocol by constructing micro-protocols designed to minimize dependency on one another. This can make the system more configurable and extensible. But, CTP does not consider partial reconfiguration after a session is established as other systems do. A main difference between DR-TCP and above systems is that in DR-TCP any functions of the protocol can be reconfigured in runtime.

STP (Patel et al., 2003b) is a protocol upgrade framework that downloads the entire protocol source code. But, the total upgrading time is much longer than that of DR-TCP because STP needs downloading time and compiling. In comparison, DR-TCP provides binary-level upgrade and also supports partial upgrade of protocol function. Therefore, it reduces
downloading time and eliminates compiling time, and consequently the total upgrading time could be dramatically reduced.

3. Recursive state machine model

A finite state machine is a quintuple \( M = (Q, \Sigma, \delta, q_0, F) \), where \( Q \) is a finite set of 'states', \( \Sigma \) is the input alphabet, \( \delta \) is a function from \( Q \times \Sigma \) into \( Q \), called the transition table, \( q_0 \in Q \) is the initial state, and \( F \subseteq Q \) is the set of final states.

In a machine, if we know the current state and transition function, the next state can be determined depending on an incoming input. Since a state machine captures the behavior of complex machine well, state machine model has been widely used in protocol modeling. However, there has been no attempt to apply the model to protocol reconfiguration because the complexity dramatically increases as the number of states increase. In this section, we will explain a finite state machine that is defined newly for TCP implementation and show this model can be reconfigured any state machine that represents other TCP variant.\(^1\)

3.1. Finite state machine for TCP

**Definition 1 (Local events based finite state machine).** A local events based Finite State Machine (FSM) is a 4-tuple \( (S, S_0, E, F) \):

- \( S \) is a finite set of states of the FSM.
- \( S_0 \subseteq S \) is a set of initial states.
- \( E \) is a finite set of \( E_s \), that is a set of local events of the state \( S_k, S_k \in S \).
- \( F(s, e) \) is a relation from the (current state, local event) pairs to the next state (i.e. \( F \subseteq S \times E \times S \)).

We introduce Local events based FSM based on TCP state transition diagram to explain TCP reconfiguration theoretically. While general finite state machine has global alphabets as inputs, each state in the TCP state diagram has different event set (or inputs) one another. Thus, \( E \) can be defined as a finite set that has sets of local events in order of state number. And, state transitions are achieved deterministically only for local events of the state.

3.2. Operations for reconfiguration

Consider a state machine \( M \) we want to reconfigure into a target state machine \( M' \). Since needing to know what transitions have to be reconfigured when migrating \( M \) to \( M' \), we define Delta Transition according to Koster and Teich (2002) as follows.

**Definition 2 (Delta Transition).** Given a finite state machine \( M = (S, S_0, E, F) \) and a target finite state machine \( M' = (S', S'_0, E', F') \), let \( T' = \{(s_x, e, s_y) : s_x \in S', e \in E_x, s_y \in F(s_x, e)\} \) denote the total set of transitions of \( M' \). Then a transition \( t_d = (s_x, e, s_y) \in T' \) is called delta transition and needs to be reconfigured in order to mutate \( M \) into \( M' \). Let \( T_d \subseteq T' \) be the set of all delta transitions.

**Example 1.** Consider the finite state machine \( M = (S, S_0, E, F) \) with \( S = (S_0, S_1, S_2) \), \( E = \{E_{S_0}/E_{S_1}/E_{S_2}\} \), \( E_{S_0} = \{e_0, e_1\} \), \( E_{S_1} = \{e_2\} \), \( E_{S_2} = \{e_3, e_4\} \) and the finite state machine \( M' = (S', S'_0, E', F') \) with \( S' = (S_0, S_1, S_2) \), \( E' = \{E_{S_0}/E_{S_1}/E_{S_2}\} \), \( E_{S_0} = \{e_0, e_1\} \), \( E_{S_1} = \{e_5, e_6\} \) shown in Fig. 1. If we want to reconfigure \( M \) into \( M' \), the delta transitions are \( T_d = \{(S_1, e_2, S_3), (S_2, e_3, S_5), (S_3, e_5, S_1), (S_3, e_6, S_2)\} \).

**Definition 3 (Reconfiguration parameter).** Given a finite state machine \( M = (S, S_0, E, F) \) and a target finite state machine \( M' = (S', S'_0, E', F') \), reconfiguration parameter \( P \) can be defined as \( (S_r, S_d, E_r, T_d) \):

- \( S_r = \{s : s \in S' \text{ and } s \notin S\} \)(i.e. \( S' - S \))
- \( S_d = \{s : s \in S \text{ and } s \notin S'\} \)(i.e. \( S - S' \))
- \( E_r = \{E_r : E \in E' \text{ and } E \notin E\} \)(i.e. \( E' - E \))
- \( T_d \) delta transition set.

\(^1\) To show feasible reconfiguration of our finite state machine model, we follow the methodology for a constructive proof in Koster and Teich (2002).
Reconfiguration parameter is required in order to reconfigure $M$ into $M'$. $S_r$ is a set of new adding states, and local events of the states is included in $E_r$. $S_d$ is a set of useless states after reconfiguration. To remove the states of $S_d$, we introduce an operation, REM(state). The function of REM(s) is to remove the state s, local events of s, and all relations from s.

**Example 2.** In Fig. 1, reconfiguration parameter P for migrating $M$ to $M'$ can be defined as $(S_r, S_d, E_r, T_d)$ with $S_r = \{ S_3 \}$, $S_d = \emptyset$, $E_r = \{ E_{s_5} \}$ and the delta transition set $T_d$ described in Example 1.

### 3.3. Algorithms for reconfiguration

In this section, we will explain reconfiguration process based on previous definitions and show that a FSM $M$ can be always reconfigured arbitrary FSM $M'$.

**Theorem 1** (Feasible reconfiguration). Given a FSM $M$ and a FSM $M'$, it is always possible to specify a finite sequence of reconfiguration steps in order to transform $M$ into $M'$.

List 1 shows the algorithm for reconfiguration, that provides a constructive proof of Theorem 1. It has two stages: (1) reconfiguring $M$ with delta transitions (steps 2–5), and (2) removing useless states and their information with REM operation (steps 6–9). Through the algorithm, a finite reconfiguration sequence $Z$ is generated to transform $M$ into $M'$. Let us explain these reconfiguration steps easily by Example 3. After all, this means that a TCP variant can be always reconfigured to any other variant based on the local events based FSM model.

**List 1. Reconfiguration Steps**

1. $n = 1$;
2. For all $t_d \in T_d$
3. $z_n = \text{delta transition } t_d$
4. $n = n + 1$;
5. }
6. For all $s \in S_d$
7. $z_n = \text{REM}(s)$;
8. $n = n + 1$;
9. }

**Example 3.** Consider the finite state machine $M = (S, S_0, E, F)$ with $S = \{ S_0, S_1, S_2, S_3, S_4, S_5 \}$, $E = \{ e_0, e_1, e_2 \}$, $\{ e_3, e_5 \}$, $\{ e_6, e_7 \}$, $\{ e_8 \}$, and transition function $F$ defined as $F(S_0, e_0) = S_1$, $F(S_1, e_1) = S_2$, $F(S_1, e_2) = S_3$, $F(S_2, e_3) = S_3$, $F(S_2, e_4) = S_4$, $F(S_3, e_5) = S_5$, $F(S_4, e_7) = S_5$ and $F(S_5, e_8) = S_4$ shown in Fig. 2(a). With the reconfiguration parameter $P = (S_r, S_d, E_r, T_d)$, $S_r = \{ S_6 \}$, $E_r = \{ e_9, e_{10} \}$, $S_d = \{ S_4, S_5 \}$ and $T_d = \{ (S_2, e_5, S_6), (S_3, e_6, S_8), (S_6, e_9, S_2), (S_6, e_{10}, S_1) \}$, the FSM $M$ can be reconfigured to $M'$ (Fig. 2(b)). Through reconfiguration algorithm, $Z$ has six steps as follows:

$$Z = \{ z_1(t_d(S_2, e_5, S_6)), z_2(t_d(S_5, e_6, S_5)), z_3(t_d(S_6, e_9, S_2)), z_4(t_d(S_6, e_{10}, S_5)), z_5(\text{REM}(S_4)), z_6(\text{REM}(S_5)) \}$$

We extend this state machine model to be recursive in that a state includes another state machine. Such a recursion simplifies a complex system by using abstract levels of recursion. Recursive state machine has a rule, “a state can be another state machine $M''$”, in addition to local event-based finite state machine’s definition.

In a state machine, generally transition comes with an action. Thus by abstracting the state transition and its behavior by object, and then delegating all behaviors to the object, state transitions can be well split into objects (Wolczko, 1992). The delegated object processes the events sent to the object without any help of parent object. Object delegation makes complex behaviors of a protocol easily abstracted by multiple levels.
In this section, we describe how the recursive state machine is applied to model functions of a protocol in details. Protocol framework and state machine objects, and their relation are depicted in Fig. 3. Protocol framework is a state machine that consists of event queue, state transition table, protocol global data – such as TCP’s TCB (Transmission Control Block), and \texttt{start()} method and objects. A state machine object consists of event queue, transition table, and a set of child state machines (objects) and \texttt{start()} method. Note that a state machine object indicates implementation of a state in a recursive state machine in Section 3. \texttt{start()} method determines which object should handle the event and manages the needed resources to actually execute the function. Note that the protocol framework reflects the highest level function on the protocol.

Events are classified into global and local events. The global event is an event that comes into the state machine object from outside like packet reception. Every state machine object reserves event number 0 and 1 for packet transmission/reception events. In contrast, the local event is an event that local state machine objects invoke depending on their semantics. So the events have to be defined by a protocol designer.

State transition table consists of current state, event, and its handling object. That is, when a state is ‘0’, and event ‘X’ occurs, and delegated state machine object is ‘obj’, the row of the table is (0, X, obj). In practice, the transition table is a hash table that has a row of (hash code, state machine object). When an event occurs, state machine object delegates its handling to another state machine object that is linked with (current state, event number) at the transition table. Actual event handling is performed by \texttt{start()} method of the object handling the event.

Protocol global data is totally dependent on the semantics of a protocol. The representative example is TCB in TCP in which all results of TCP behavior are stored.
A key advantage of applying recursive state machine model to the functions of a protocol is that the inter-dependency among protocol functions is simplified through object delegation. In other words, protocol functions share many data structures, which make dynamic reconfiguration extremely difficult. By using state machine objects, each function of a protocol is clearly isolated from other objects so that interface between objects is clear and consistent (by \textit{start()} method). A detailed description for TCP is given in the next section.

5. Structuring TCP in recursive state machine model

In this section, using the concept of the recursive state machine, we restructure TCP by classifying protocol framework and state machine objects. First of all, we consider the original state diagram of TCP in Fig. 4 (called, level 0).

To implement TCP Reno, we have to define a framework and state machine objects that perform each protocol function of TCP. As we mentioned in previous section, TCP framework is a state machine object with TCB. And, each function is wrapped by state machine objects such as tcpClosed, tcpListen, tcpSynsent, tcpSynrecvd, tcpEstablished, tcpFinwait1, tcpFinwait2, tcpClosing, tcpTimewait, tcpLastack, tcpClosewait in Fig. 4 (each object name is made by prefix “tcp” and each state name). When being created, the state machine object automatically calls its constructor.

Now calling \textit{buildTTable()} method in the constructor, the framework creates the transition table that includes state machine objects, state information, and event numbers. And then invoking \textit{start()} method, the framework waits for events in the event loop.

When the system receives a packet, interrupt occurs, and its handler puts PKTRCVD (event number 0), one of global events, into the event queue of tcpMachine. tcpMachine’s \textit{start()} method makes the proper event depending on the semantics of TCP, and the framework delegates the event handling to a state machine object that is linked with its hash code in the transition table. For example, let us consider the TCP connection process. As we can see the state diagram in Fig. 4, tcpMachine is first created. When PASSIVEOPEN event occurs, tcpListen calls the \textit{start()} method. The \textit{start()} method sends SYN+ACK packet to peer if the state machine object receives PKTRCVD event and the received packet has SYN flag with its header. And then, by invoking the SYNACK event, tcpListen object delegates further processing to tcpSynrecvd object. tcpSynrecvd object responds on PKTRCVD event, and if the received packet is ACK, the object acks.

![Fig. 4. TCP state diagram (abstract level 0).](image)
tcpEstablished object to process further events. tcpEstablished object is another conceptual state machine that handles TCP data transmission/reception, congestion control, retransmission, and so forth.

Therefore the tcpEstablished object has a set of events that may happen and states, and Fig. 5 shows its state diagram. As we can see in Fig. 5, in the state machine object, general packet transmission is performed by tcpTransmit object. But if the retransmission timer is expired or the machine receives zero window advertisement, tcpRetransmit and tcpPersist objects are activated respectively. Also, when the object receives three duplicated ACK, tcpFastrtxrecovery object gets the control.

Because the functions of tcpTransmit object are slow start and congestion avoidance, the object can be further subdivided into two state machine objects. However, to focus on the implementation model, we do not further extend the example.

6. Implementing TCP in recursive state machine model

In this section, we implement TCP based on the state machine model and TCP structure in previous section. The only difference between the conceptual model and the implementation is that we do not use event queue (it is for this reason that TCP functions behaves synchronously). Because each state machine does not queue the event messages, when an event occurs, the next event is not handled until processing the event completes. And, to do communication, we use raw socket interface (Stevens, 1998) since our TCP is implemented in user level for the sake of convenience of testing and debugging. Implementation environment is Linux, and we use gcc 3.2.2 compiler (FSF, 2002).

As shown in Fig. 6, DR-TCP takes a role of transport protocol at user level generating TCP header manually by raw socket interface, and using setsockopt function with IPPROTO_IP and IP_HDRINCL flag, we can set the protocol field of IP header to 6 that indicates the protocol of transport layer is TCP. Therefore, packets generated by DR-TCP are considered as normal TCP packets in the receiver side. As a result, DR-TCP can communicate with other normal TCP implementation. But, if DR-TCP coexists with normal TCP in same host, some kernel modifications are required in order to distinguish incoming DR-TCP packets from normal TCP packets. We will provide a simple solution of this problem in Section 6.3.

6.1. Downloadable and reconfigurable TCP

The basic goal of our TCP implementation is to make TCP be partially upgraded. And the ultimate goal of our trial is to make TCP be reconfigured through the partial upgrade. Thus, we name the implementation DR-TCP, which is the abbreviation for Downloadable and Reconfigurable TCP. In this section, we begin explanation by comparing DR-TCP’s each component with the conceptual model.
6.1.1. Defining state machine object

Conceptually a state machine object consists of event queue, transition table, and an execution interface of the state machine, \texttt{start()} function. In our implementation, except for removing the event queue, we implemented all the components as in List 1. A state machine object is created and destroyed by \texttt{smCreate()}\texttt{smDestroy()} methods. We need to give attention to the structure of \texttt{start()} method that operates the state machine object. The role of \texttt{start()} interface is to handle global/local events. List 3 shows the skeleton of formal \texttt{start()} interface. When \texttt{start()} function receives the control, the method first checks the global events that current state machine object has to handle, and then, if it has, \texttt{start()} does a proper operation. In List 3, we can see that the method processes the two global events, PKTSENT and PKTRCVD, which are generated when a packet is sent and received. First of all, the control jumps to PKTOUT or PKTIN depending on the occurred global event, and the processing of such global event leads to generate some local events. And then, the control is passed to EXEC_MACHINE label, where the local events of the state machine object are processed. The control flow within the state machine is simple. That is, every state machine object follows the control sequence of \texttt{start()} \rightarrow handling global events \rightarrow processing the packet and generating the local event \rightarrow handling local events, and usually the local event processing are delegated to another state machine object. The actual meaning of delegation in this paper is to shift current active state machine to another state in which the next events would be processed. And, the \texttt{actSMachine} field in List 2 keeps track of the pointer of current active state machine object. To explain these process steps easily, for instance, let us assume that the active state machine object is tcpListen. When the system receives SYN request, the framework of DR-TCP invokes \texttt{start()} function of current state machine, tcpListen, with PKTIN label of tcpListen’s \texttt{start()}, and as a result, local event SYNSENT occurs. Consequently, processing the local event in EXEC_MACHINE leads to transit the active state machine object to tcpSynrecvd, and the final ACK packet of three-way handshaking will be processed in tcpSynrecvd state machine.

\texttt{pref} and \texttt{fref} always indicate parent state machine and framework objects respectively. The reason to need the pointer of parent state machine object is that the transition is performed by using the parent’s transition table. And, the pointer of the framework is used to access the global data of the protocol, TCB.

List 2. State machine structure

```c
#define SMACHINE char cname_[STRLEN];
void (*start)(struct Framework*, struct Packet*, int tcpEvent);
void (*buildTTable)(struct SMachine*);
struct HashTable* TransitionTbl_;
SMachine* actSMachine; // active state machine indicator
SMachine* pref; // parent state machine
SMachine* fref; // framework state machine

typedef struct SMachine {
   SMACHINE;
} SMachine;
```

Fig. 6. DR-TCP implementation architecture.
6.1.2. Defining TCP framework and protocol component

As we mentioned in previous sections, framework is another state machine object that consists of basic state machine, the global data of protocol, and start() method. Since the target protocol is TCP, our framework contains TCB as the global data (see List 4).

Protocol component consists of a set of state machine objects that implement each function of the protocol and a header that indicates the position of machine in the framework. Component header consists of three fields: the first field is component ID, the second field is the number of key values, and the last field is key value list. Conceptually the number of key values is not limited, but, for the sake of convenience of implementation, we restrict the number to 256 (Fig. 7).

The way to find a proper position of the component is easy. The framework applies key #1 to the transition table and again repeats it for the next key by the number of key values. As the result of repeated operation, we are able to find the proper transition table of the state machine object. Fig. 8 draws search process for the reconfiguration using the key values, to replace the state machine object by the downloaded object.

6.2. DR-TCP operation

We present the operation of DR-TCP following the code execution sequence with the explanation of TCP framework member functions. For the sake of convenience, all capital symbols denote conceptual state (for example, ESTABLISHED), and symbols with ‘tcp’ and only first capital letter mean the instance of state machine (tcpEstablished). Also, we will use SM as acronym of state machine object.

tcpfBuildTTable() function that makes transition table of TCP framework creates state machine objects of CLOSED, LISTEN, ESTABLISHED, and so forth in Fig. 9, and then, the function set TCP framework’s actSMachine to a SM that handles initial state of TCP (So, in case of active open, the initial state is CLOSED while, in case of passive open, the initial

### List 3. Start() interface in implementation skeleton

```c
// Global Event Handling
switch (G_Event)
    case PKTSENT: goto PKTOUT;
    case PKTRCVD: goto PKTIN;
    ...
}
...
PKTOUT:    // Outgoing packet processing
P KTIN:    // Incoming packet processing
...
EXEC_MACHINE:
    // Local event handling
```

### List 4. TCP Framework data structure

```c
typedef struct Framework
    SMACHINE;
    // from here, global protocol data
    struct tcb * ftcb; // Transmission Control Block (TCB)
} Framework;
```

Protocol component consists of a set of state machine objects that implement each function of the protocol and a header that indicates the position of machine in the framework. Component header consists of three fields: the first field is component ID, the second field is the number of key values, and the last field is key value list. Conceptually the number of key values is not limited, but, for the sake of convenience of implementation, we restrict the number to 256 (Fig. 7).

The way to find a proper position of the component is easy. The framework applies key #1 to the transition table and again repeats it for the next key by the number of key values. As the result of repeated operation, we are able to find the proper transition table of the state machine object. Fig. 8 draws search process for the reconfiguration using the key values, to replace the state machine object by the downloaded object.

### Fig. 7. Component header structure.
state is LISTEN). If TCP is active open, framework’s `start()` function calls CLOSED SM’s `start()` function, which transmits first packet of three-way handshake, SYN. After this, receiving SYN+ACK from the peer, SYSENT SM transmits ACK and delegates further processing to ESTABLISHED SM (this phase completes three-way handshake).

Actual exchange of TCP data starts from ESTABLISHED. Fig. 9 draws the control flow when `tcpEstablished` object performs data exchange. In this situation, the control can flow to three ways. First, in normal packet transmission, the control runs in `Tcpwrite()` → `tcpEstablished` → `start()` → `tcpTransmit` → `start()` → `sendSegment()`.

Second, in the packet arrival, the control flows in interrupt → `recvSegment()` → `tcpEstablished` → `start()` → `Tcpread()`. And Last, when the timeout occurs, the execution goes in timeout by signal → `tcpEstablished` → `start()` → `tcpRetransmit` → `start()` → `sendSegment()`.
Observation of the control flow says that there is a rule: when global events occur, invariably \texttt{start()} function of the activated state machine is called. In case of Fig. 9, since currently activated SM is tcpEstablished, always tcpEstablished- > \texttt{start()} is called first. Afterward, the flow would be different depending on delivered events. An important thing is that tcpTransmit and tcpRetransmit are child objects of tcpEstablished, and tcpEstablished's active state indicator is initialized to point to tcpTransmit. That is, when tcpEstablished receives PKTSENT event, tcpTransmit- > \texttt{start()} is called. And also, when TIMEOUT event occurs, tcpEstablished's indicator points to tcpRetransmit by the state transition. In case of PKTRCVD event, PKTIN section of tcpEstablished is executed regardless of type of child SM.

6.3. Structural implementation issues

Fig. 10 draws our implementation architecture from a different point of view with Fig. 6. As we can see the figure, DR-TCP coexists with application in a process space (process #1) as the shape of library. On the contrary, reconfigured state machines- SM #1' and SM #2' - and the reconfiguration loader are located in another process space, process #2. It is because that we implement DR-TCP in the user level, and this design needs to share memory area between the loader and DR-TCP. So, we used shared memory, one of the traditional UNIX IPCs (Hall, 1997). If DR-TCP and the loader are implemented in the kernel level, such IPC mechanism would not be required. This section addresses some problems from the implementation structure of Figs. 6 and 10.

6.3.1. How to allocate state machine objects in shared memory

If \texttt{malloc()} is used to allocate state machine objects for DR-TCP, we cannot perform the reconfiguration even when the replaceable SMs is loaded into the shared memory area. It is because the memory blocks to be returned by \texttt{malloc()} exist in the heap area. In other words, when DR-TCP uses \texttt{malloc()} in Fig. 10, because the state machine objects are allocated in the heap, the protection domain of process #1, the reconfiguration loader in process #2 cannot access the objects. That is, due to protection domain, the loader cannot copy the replaceable SMs into the address space of DR-TCP, and it also cannot modify the transition tables of DR-TCP. To solve this problem, we implemented a simple memory allocator named DR-allocator that manages shared memory area. We will explain this allocator with the reconfiguration loader in the next section.

6.3.2. How to modify the kernel to communicate with other TCP implementation

The packet multiplexing mechanism of Linux kernel is simple. First of all, a protocol of transport layer depending on the protocol field of IP header is selected. If the field is 6 that indicates TCP, \texttt{tcp_v4_rcv()} function is called. The function checks whether the session identifier of the packet exists in socket list or not (session identifier consists of {source ip address, source port, destination ip address, destination port, network interface}). And then, if there is not the same identifier, the kernel discards the packet. With the same reason, when the kernel receives a packet with the protocol field of 255 (255 indicates raw socket), after checking the session identifier, the packet would be processed by raw socket or discarded.

![Fig. 10. Object sharing for the reconfiguration of DR-TCP.](image-url)
To summarize the problem, we need to consider a case in which DR-TCP is sender, and Linux TCP is receiver. After completing TCP processing, DR-TCP replaces the protocol field of 255 by 6 and transmits the packet. Since the protocol field of received packet is 6, the receiver kernel processes the packet by calling \textit{tcp\_v4\_recv()} function. The receiver kernel returns ACK packet by TCP semantics, and, because the protocol field of this packet is 6, not DR-TCP but Linux TCP receives the packet in the sender side. This end-to-end communication comes to fail, for Linux TCP cannot find the correct session identifier.

To solve this problem, we modified \textit{tcp\_v4\_recv()} function that performs the TCP session identification in Linux kernel. A different mechanism from the original function is that the function does not directly discard the packet even when the kernel cannot find the session identifier. That is, the modified function again performs the packet multiplexing phase of raw socket as if TCP receives the packet with protocol field of 255. Through the stage, finally DR-TCP can come to receive ACK packet, and, it help the protocol correctly operated.

6.3.3. How to handle the packet arrival interrupt

When the packet is received, an interrupt occurs, and then, its interrupt handler performs proper processing depending on its protocol. A question from our implementation model is that implemented protocol code runs in user-level while interrupt handler runs in kernel-level. For two processing phases to be synchronous, separation of execution code is not desirable. A intuitive solution is that the interrupt handler directly calls DR-TCP’s \textit{recvSegment()} function, which is called upcall (Clark, 1985). However, calling \textit{recvSegment()} function directly makes the interrupt handler too heavy. So, we imitated the upcall mechanism by using the process signal and Linux bottom half mechanism.

The packet reception process of DR-TCP is as follows: When the packet arrival interrupt occurs, the interrupt handler marks that bottom half processing is required. Through the bottom half processing, when \textit{tcp\_v4\_recv()} function gets the control, the function checks whether the packet is DR-TCP’s or not. If it is, \textit{tcp\_v4\_recv()} function sends signal to DR-TCP application. After this, when the kernel completes other system jobs, and DR-TCP process is called by process scheduler, first of all the packet reception is performed by calling \textit{recvSegment()} function. Note that \textit{recvSegment()} function is the first initiator of \textit{Tcpread()} function.

7. Reconfiguration loader

To actually reconfigure DR-TCP, two system level supports are needed. One thing is a new memory allocator, and the other thing is dynamic linking loader.

When the reconfiguration loader loads new protocol components after downloading stages, it has to know the memory address of state machine objects of both DR-TCP and downloaded TCP in order to modify transition tables. As shown in Fig. 10, we implement a simple memory allocator named DR-allocator, and its main role is to keep track of SM map of DR-TCP and provide object information to the loader. Our allocator uses block header algorithm (Crowley, 1996), and to identify the object type of allocated area, we only add a simple field to the formal block header. Using this identifier, the loader can accurately get the proper DR-TCP SMs.

Another important function of the reconfiguration loader is to dynamically rearrange the symbols of protocol component that are also used in DR-TCP. For instance, let us consider the case that the downloaded component references DR-TCP’s \textit{tcpsend()} function in the example of TCP Westwood (of course, if the component contains in-house function of TCP packet transmission, this progress is unnecessary). The reconfiguration loader loads the SMs of protocol component into a suitable memory area, and it rearranges the unresolved references with currently 0x0. The rearrangement is calculated based on the following Eq. (1).

\[
\text{call parameter} = \text{relocation base address} + 4 + r_{offset of symbol} - \text{absolute address of symbol}
\]

Here the reason why Eq. (1) contains ‘+4’ is that call and jump instruction use relative addressing, so the value in the instruction is the difference between the target address and the address of the byte following the instruction itself. The reverse, variables and \textit{.rodata} symbols are rearranged based on the following Eq. (2).

\[
\text{symbol address} = \text{relocation base address} + r_{offset of symbol} - \text{absolute address of symbol}
\]
fined symbols related to functions of DR-TCP and loads state machines of the component into address space of DR-TCP (i.e. shared memory between DR-TCP and the reconfiguration loader). Otherwise, the component should be downloaded before linking and loading stages. Finally, the DR-TCP framework can search the proper transition tables according to hash key lists in the component header to replace old state machine by new one.

If the requested protocol component is already included in DR-TCP source tree, loading stage for the protocol can be done by just creating state machine objects using DR-allocator without assistant of the reconfiguration loader.

8. Experiments and analysis

In this section, we perform experiments of two types. The experiments of first type are to validate the reconfiguration of DR-TCP, and the experiments of second type are to show the effective upgrade of DR-TCP. In all experiments, maximum segment size (MSS) is fixed by 536 bytes.

8.1. Validating reconfiguration of DR-TCP

To validate reconfiguration of DR-TCP, we perform two experiments over Internet in terms of feasibility and overhead. First experiment is to observe reconfiguration process from DR-TCP to TCP Westwood. The study is progressed by two phases. In the first phase, by TCP, the connection is established and the session starts. And, after 20 RTTs, TCP is dynamically reconfigured into TCP Westwood.

TCP Westwood determines ssthresh by estimating end-to-end bandwidth based on ACK arrival rate after the timeout occurs. Therefore, when comparing with TCP Reno, we can summarize the protocol’s unique mechanisms as follows: (1) bandwidth estimation mechanism that uses ACK arrival rate and (2) ssthresh calculation mechanism that uses the estimated bandwidth. Thus, in order to reconfigure DR-TCP into TCP Westwood, state machines that we have to modify are only three SMs: tcpEstablished, tcpRetransmit, and tcpFastrtxrecovery. It is because tcpEstablished object handles ACK processing, and tcpRetransmit and tcpFastrtxrecovery objects perform congestion avoidance in basic DR-TCP. For TCP Westwood operation, tcpEstablished object adds bandwidth estimation function to the basic facility, and tcpRetransmit and tcpFastrtxrecovery objects replace the original ssthresh calculation mechanism with new one based on the bandwidth estimation.

Fig. 12 shows the variation of cwnd during reconfiguring DR-TCP into TCP Westwood. The difference of two protocols is ultimately how each protocol responses on the timeout event, and the change of cwnd and ssthresh shows effectively the difference. As we can see in Fig. 12, in the first phase, ssthresh is reduced by the half of cwnd when the timeout occurs or three duplicated ACKs arrive, which is by TCP standard’s mechanism (Postel, 1981). On the contrary, with same events in the second phase, ssthresh is elaborately adjusted by not just the half of cwnd at the occurrence of congestion but the calculation of TCP Westwood. The vertical dash line in Fig. 12 indicates the time of the reconfiguration completion, and dark/white circles are marked with reno’s and westwood’s ssthresh respectively. Note that reno’s ssthresh after the reconfiguration is artificially drawn for the sake of comparison. Consequently we can confirm that DR-TCP is reconfigured into TCP Westwood in run-time without the break of session.

Second experiment is to measure the reconfiguration overhead. To clock delayed time due to process reconfiguration precisely, we use Time Stamp Counter (TSC) timer, which is a high resolution timer supported by Intel Pentium processor (Kris Kaspersky, 2003).

Table 1 shows the reconfiguration overhead without upgrade process. We divide the reconfiguration process into two steps, which are: (1) creating SM objects and (2) searching transition tables since the first step is not required if the objects
are already in the local repository. The target SMs are tcpEstablished, tcpRetransmit, and tcpFastrtxrecovery in this experiment. When only tcpEstablished SM is reconfigured, the total reconfiguration overhead is 2.665 ms, and marginal overhead times are 1.265 ms and 1.081 ms, respectively as other two SMs are reconfigured additionally. Even though the marginal overhead of each state machine can be different a little according to the number of hash keys to be referred, the total overhead time seems to be increased linearly. That is, the reconfiguration overhead is dominated by the number of SM. As a result, when considering most TCP variants can be reconfigured by three or less state machines from our experience, we can say that the reconfiguration overhead of proposed TCP structure is very low.

8.2. Comparing DR-TCP with existing upgrade solutions

A distinctive feature of DR-TCP is that it is possible to upgrade partial function of binary level. Compared with the source-level solution, Self-spreading Transport Protocol (STP), our scheme’s downloaded code is extremely small, and it does not require any compilation phase and any specific session migration. Moreover, the total upgrade time is dramatically short. This section performs the experiment that can show such advantage of DR-TCP compared with STP.

We can summarize the process of STP upgrade as follows:

- The whole TCP source code is downloaded.
- The code is compiled, linked, and then created as a complete execution image.
- The loader loads the created image and migrates the session.

DR-TCP’s upgrade process is summarized as follows:

- The binary code of partial function is downloaded.
- The reconfiguration loader dynamically links the symbols of binary code with the framework’s symbols.
- The loader loads the binary code into a proper area based on the header information.

Since it is difficult to compare DR-TCP with STP absolutely, we referred the results in Patel et al. (2003b) as upgrading time of STP shown in Table 2. Also, for the sake of experimental convenience, we ignored the overhead of session migration.

<table>
<thead>
<tr>
<th>Number of SM</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating objects</td>
<td>0.679 ms</td>
<td>0.855 ms</td>
<td>1.122 ms</td>
</tr>
<tr>
<td>Searching tables</td>
<td>1.986 ms</td>
<td>3.075 ms</td>
<td>3.889 ms</td>
</tr>
<tr>
<td>Total</td>
<td>2.665 ms</td>
<td>3.930 ms</td>
<td>5.011 ms</td>
</tr>
</tbody>
</table>

Fig. 12. Variation of ssthresh before/after the reconfiguration.

In Patel et al. (2003b), they implemented a prototype of the STP framework in the FreeBSD Kernel version 4.7 and measured shipping, compiling, and loading time of NewReno STP on an 850 MHz Intel Pentium III with 512 MB of SDRAM over 2 Mbps link.
tion. The downloading time of DR-TCP was measured under the assumption of 2 Mbps wireless link and non-congestion. The compiling and loading time was measured with the machine that has Pentium 1.5 GHz CPU, 256 MB RAM, 64 KB L1 cache, and 1 MB L2 cache.

Table 2 shows the comparison of DR-TCP’s partial upgrade and full source-level upgrade (STP) when reconfiguring (or upgrading) TCP Reno into NewReno.

Fig. 13 shows the comparison of the total upgrade time of two schemes.

As we can see in Table 2 and Fig. 13, the upgrade time of DR-TCP is very small compared with STP upgrade. In addition to that, a remarkable thing is that, even though the overhead of session migration is removed in the experiments, our scheme shows a dramatic improvement in the performance. Because the source-level upgrade requires a specific mechanism for the session migration, actual difference between the real and the experimental environments would be larger (Note that DR-TCP needs not any mechanism for the session migration).

In Table 2 and Fig. 13, the most dominant factor of the upgrade overhead of STP is the compiling time. Actually, the compiling time of STP is about 87% of total upgrade overhead, and it requires development tools for compilation while most hand-held devices are not provided with a compiler. On the other hand, the compiling time of DR-TCP is 0 since protocol component of DR-TCP is binary code. Thus, we need not compile the code again, and it means that DR-TCP has two advantages: (1) reduce upgrade overhead significantly and (2) require no compiler (i.e. our solution is suitable to be deployed in mobile devices). As a result, we can confirm that removing compilation phase dramatically reduces the total upgrade time.

Fig. 13 also says that the most dominant factor of DR-TCP is the downloading time. Although the downloading time of DR-TCP is about 97% of total upgrade overhead, it takes still less than the downloading time of STP because DR-TCP uses 10KB-binary partial code as an upgrading code whereas STP requires 86KB-full source code as shown in Table 2. A noteworthy point is that the downloading time is seriously affected by the network bandwidth and condition. We believe that the downloading time can be reduced as the network bandwidth increases, and at least our solution would be faster than the source level one.

We note that there are binary-level upgrade solutions such as LKM (Linux Kernel Module), RPM (RPM Package Manager). However, upgrading method of those solutions is rather static so that new components should be installed manually by a user. In addition, they often require restarting the application that causes loss of connectivity. We summarize features of existing upgrade solutions and DR-TCP in Table 3.

<table>
<thead>
<tr>
<th></th>
<th>DR-TCP</th>
<th>STP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downloaded code size</td>
<td>10.428 KB</td>
<td>86.67 KB</td>
</tr>
<tr>
<td>Downloading time</td>
<td>0.271 s</td>
<td>1.0 s</td>
</tr>
<tr>
<td>Compiling time</td>
<td>0.0 s</td>
<td>10.0 s</td>
</tr>
<tr>
<td>Linking and loading time</td>
<td>0.0074 s</td>
<td>0.5 s</td>
</tr>
<tr>
<td>Total upgrading time</td>
<td>0.2784 s</td>
<td>11.5 s</td>
</tr>
</tbody>
</table>

Fig. 13. Comparison of total upgrade time of two schemes.
9. Conclusion

In this paper, we re-structure TCP Reno to enable reconfiguration of its functions and encapsulated each function unit by a state machine. Based on our recursive state machine model, we show that it can be reconfigured to any TCP variants theoretically. Then, we implement the re-structured TCP and name the protocol DR-TCP. DR-TCP can easily download each function of TCP, and, through the reconfiguration loader, it can dynamically reconfigure the functions. Through the experiment of reconfiguring DR-TCP into other TCP variants, we show that DR-TCP not only dramatically reduces the size of downloadable protocol functions but also makes protocol reconfiguration extremely simple. Moreover, based on several real experiments in Internet, we demonstrate that the reconfigured TCP Westwood works well.

DR-TCP is a result of our effort to make reconfigurable protocol stack. Perceiving that almost every protocol can be modeled as state machine, we apply recursive state machine to TCP implementation. And, through DR-TCP, we believe that partial upgrade of a protocol can be done in actual deployment.

Acknowledgement

This work was supported by grant No. R01-2004-000-10588-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

References

IEEE Transactions on Parallel and Distributed Systems 10 (6), 600–612.

Jae-Hyun Hwang received the B.S. degree in computer science from Catholic University, Seoul, Korea, in 2003 and the M.S. degree in computer science from Korea University, Seoul, Korea, in 2005. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interests are in network protocol design and kernel networking.
Jin-Hee Choi received the B.S., M.S., and Ph.D. degrees in computer science from Korea University, Seoul, Korea, in 2000, 2002, and 2006, respectively. Currently he is with Telecommunication R&D Center, Samsung Electronics. His current interests are in kernel networking and mobile software platform.

Se-Won Kim received the B.S. and M.S. degrees in computer science from Korea University, Seoul, Korea, in 2004 and 2006, respectively. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interests are in the temperature-aware computer architecture and kernel networking.

Chuck Yoo received the B.S. degree in electronic engineering from Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science in University of Michigan. He worked as a researcher in Sun Microsystems Laboratory, from 1990 to 1995. He is now a Professor in Department of Computer Science and Engineering, Korea University, Seoul, Korea. His research interests include high performance networks, multimedia streaming, and operating systems. He served as a member of the organizing committee for NOSSDAV 2001.