Signalling Protocol for P4 (SPP4), version 1.0

Ilija Hadžić
Distributed Systems Laboratory, University of Pennsylvania
ihadzic@ee.upenn.edu

June 26, 1998

Abstract

This document describes SPP4 v.1.0, a signalling protocol implemented on Programmable Protocol Processing Pipeline (P4) platform. Protocol has been implemented on P4 v.1. It is expected that P4 v.2 will use a modified version of this protocol to support new features.

Current version of the signalling protocol allows P4 boards to synchronize their actitivies. Protocol Boosters model is supported and it is assumed that the mechanism modules are already available at each of the P4 enhanced nodes. Providing the mechanism modules from by the end user is not supported yet.

The signalling protocol is specified to support non-transparent boosters (i.e. boosters that require deboosters). If only transparent boosters are used (i.e. those that can operate without debooster), some of the features may be unnecessary. The existing implementation can work with transparent boosters but it will have more overhead than necessary.

This document covers protocol specification, interaction between signalling protocol and policy modules and software architecture of current implementation. The purpose of this document is to explain, not to formally specify the SPP4.

1 Overview

The SPP4 implementation described in this document runs on P4 platform[HS97]. Although this document concentrates on P4 specific implementation, higher level concepts can be generalized to any implementation of Protocol Boosters. Possible extensions of P4 to fully support the Active Network [TSS+97] acceleration is not discussed here.

SPP4 consists of three modules. On the highest level is the signalling protocol, which is independent of the particular platform and does not in general require the P4 platform. It can be incorporated in any Protocol Boosters implementation. This documents explains the operation of SPP4. A set of messages exchanged between boosters and procedures followed are described here.

Second element is the pool of policy modules which are booster-specific. Their purpose is to determine if the particular booster should be used. Different policy modules can be used to make the policy decisions in on and off state. Policy modules are implemented as the set of functions which are periodically called to check the policy decision. Each known booster has the policy module associated with it.

The third element is the platform specific device driver. Policy modules and signalling protocol both use `ioctl` system calls to communicate with the device driver. Device driver controls the P4 hardware and maintains the data structures describing the state of the board. If the SPP4 was to be ported for non-P4 platforms, device driver would implement a virtual device seen by the higher layers of the SPP4.

2 Device Driver

This section describes the functions of the device driver seen by the SPP4. The purpose is to provide enough information for understanding the signalling protocol. The actual implementation of the device driver is out of the scope of this document.

Two main functions of the device driver are managing the FPGA devices and handling signalling messages. Device configuration and its removal from the pipeline chain is done on demand from the user space program, but the device driver is free to choose the actual physical device.

Handling the incoming signalling messages is done in the interrupt service routine, and the `ioctl` function is used to request the transmission of the signalling message. We now describe the system calls needed for understanding the operation of the SPP4. We assume that the reader is familiar with general principles of device driver implementation. More information on this topic can be found in [BBD+96]

2.1 write

The `write` system call is reserved for device downloading. The user space program passes the pointer to the data structure that contains the location in the pipeline chain where the device should be inserted, unique configuration identification number, the flag indicating if the function should be only downloaded or downloaded and switched in, and the configuration data.

The device driver maintains the list of active devices and free devices. Active device is the one which has been configured and is operating in the pipeline chain. Free device is not in the pipeline chain and it can be either blank or configured, but it does not take part in data processing. Free device can be overwritten by new configuration at any time.

If the user space program requests the downloading only, the driver will download the device without switching it into the pipeline chain. Device will remain configured without doing the actual processing. The purpose of this preloading is to allow the policy module to (if possible) predict the need for the booster and configure in advance. Preloaded device stays in the list of free devices. Further in the text, we refer to this operation as `LOAD`. If the preloaded device exists when the `LOAD` is requested, the command will be ignored.

If the user space program requests download with the function switch-in, the device will be inserted into the processing chain. We refer to this operation as `LOAD&SWITCH`. After the device has been switched in, it becomes active.

If the preloaded device is found in the list of free devices, the device is simply switched in. This operation is very fast and it is comparable to the length of few bus cycles. If none of the preloaded devices in the list of free devices has the appropriate configuration, the device driver seeks for a free device and configures it. This operation is slow since downloading the FPGA device is a lengthy process. If the free device needs to be downloaded, the priority is given to a blank device because in this case downloading does not require overwriting the existing preloaded device. If no blank device is available, one of the preloaded devices must be overwritten. Studying the effects of various replacement algorithms to the agility of the network infrastructure is the objective of the future research. At this stage, device driver chooses the first device in the list. Implementation of more sophisticated replacement algorithms is expected in future versions.

To summarize, `write` system call can be invoked with two options: `LOAD` and `LOAD&SWITCH`. The purpose of the `LOAD` is to request the device advance configuration in order to increase the probability of finding a configured device when needed. The purpose of the `LOAD&SWITCH` is to insert a function in the pipeline chain and if necessary, the device will be configured prior to switching it in.

2.2 ioctl

SPP4 uses `ioctl` commands to transmit and receive signalling messages, remove a device from the pipeline chain and read the status of the device. In this section we describe the `ioctl` commands to the level of detail necessary for understanding the signalling protocol. Since the goal of this section is to provide prerequisites for describing the SPP4 protocol, only selected commands are described:

- `P4_GET_CMD` reads the received signalling message
- `P4_SEND_CMD` sends the signalling message
- `P4_RD_PE` reads the FPGA device status
- `P4_RM_PE` removes the processing element from the pipeline chain

Received signalling message is checked for bit errors and stored in the internal buffer of the P4 board. If the message passes the CRC check, an interrupt is generated. The device driver interrupt service routine reads the message and stores it in the internal buffer.

When the user space program executes the `P4_GET_CMD` command, the signalling message is passed from the receive buffer in the kernel space to the user space. The return value of the system call will indicate if the data has been successfully read or if the receive buffer is empty.

When the user space program wants to send the signalling message, it will issue the `P4_SEND_CMD` command and pass the content of the signalling message to the device driver. Signalling message is protected from errors with the CRC which is calculated at the output interface of the P4.

The status of individual processing element can be read using the `P4_RD_PE` command. In SPP4 it is used by policy modules. For example an FEC policy module can program one of the FPGA devices to monitor the link state and read the status to collect the information necessary to make policy decisions.

Processing element can be removed from the pipeline chain by issuing the `P4_RM_PE` command. The FPGA device is switched out, but the configuration is not cleared. Removed processing element becomes a free device and can be either reused with the old configuration or overwritten with the new configuration.

2.3 select

The `select` system call returns if the signalling cell is available, that is if the receive buffer is not empty. If the receive buffer is empty, a call to `select` will block the calling process, that is the main thread of the SPP4.

3 Policy Module

A policy module is a booster specific function which decides if the booster should be inserted, removed or the current configuration should remain unchanged. The code executed by the policy module depends on a particular booster. Policy module may delegate some of its functionality to processing elements on P4.

For example, a policy module associated with the FEC booster may program the FPGA device on P4 board to count good and bad packets and use this information to make policy decisions. When the policy module utilizes the FPGA device on the P4 board it will operate in the polling mode (i.e. periodically read the device status).

After execution, a policy module returns the status code which can take one of the three values:

- 0: operate without the booster (switch the booster out)
- 1: remain in current state
- 2: operate with the booster (switch the booster in)

Policy module can make the decision only for the associated booster and different policy module can be used for different booster states (i.e., one policy module can be used to switch the booster in if the protocol is operating in non-boosted mode while the other policy module can be used to switch the booster out if the protocol is operating in boosted mode).

4 Message Format

Signalling message consists of four fields as shown in Figure 1. The first field is the command code (CMD). Semantics of the message is carried in this field. The second field (DST) identifies the booster or debooster to which the message is sent and it is called the destination or the target field. The third field, source field, (SRC) identifies the booster or debooster which has sent the message. The last field (CRC) is for bit error checks. This field is calculated in hardware by the P4. In this version of SPP4, all fields are 8 bits wide. Extensions may be expected in future versions of the protocol.

CMD	DST	SRC	CRC

Figure 1: Signalling Message Format

Following values are allowed for command field:

- `PREPARE`: Request that the remote P4 board download the booster or debooster without switching it in. This message can be sent by either booster or debooster, and it is used for configuration preloading.
- `BOOST`: Request that the remote P4 board download the booster or debooster and start the synchronization procedure for switching to boosted mode. This message can be sent by either booster or debooster, but the booster will normally skip it and send the `START` message immediately.
- `START`: Request that the boosting starts. This message can be sent by the booster only and it marks the point in time when the boosted mode of communication starts.
- `STARTED`: Indicates that the boosting has started. This message is sent by debooster in acknowledgement to `START` message.
- `ABORT`: Request that the boosting stops. This message may be sent by either booster or debooster.
- `IDLE`: Indicates that the boosting has stopped. This message is sent in response to `ABORT` message.

Messages with the unknown command field will be ignored.

Logically, message is represented as `CMD(DST,SRC)`, which means: "Command `CMD` sent from source `SRC` to destination `DST`". Message whose destination code does not represent a known booster will be ignored.

Signalling message is encapsulated in the single ATM cell and is transmitted over the same virtual circuit as the data stream. First four bytes of the cell payload are used. To distinguish the signalling cell from user data and OAM cells, the reserved combination of payload type indicator (111) is used [ATM94].

5 Software Architecture

The software architecture of SPP4 is shown in the Figure 2. Device driver runs in kernel space and all other elements are implemented in user space. Main thread demultiplexes signalling messages sent from and to supervisory threads which implement the signalling protocol. Policy modules are used by supervisory threads to make policy decisions.

The main thread is blocked on `select` until the signalling cell arrives or the `select` timer expires. Timer is provided as the safety feature, so that the system does not stay blocked indefinitely. After it wakes up, the main thread will read all signalling messages from the device driver's buffer and deliver them

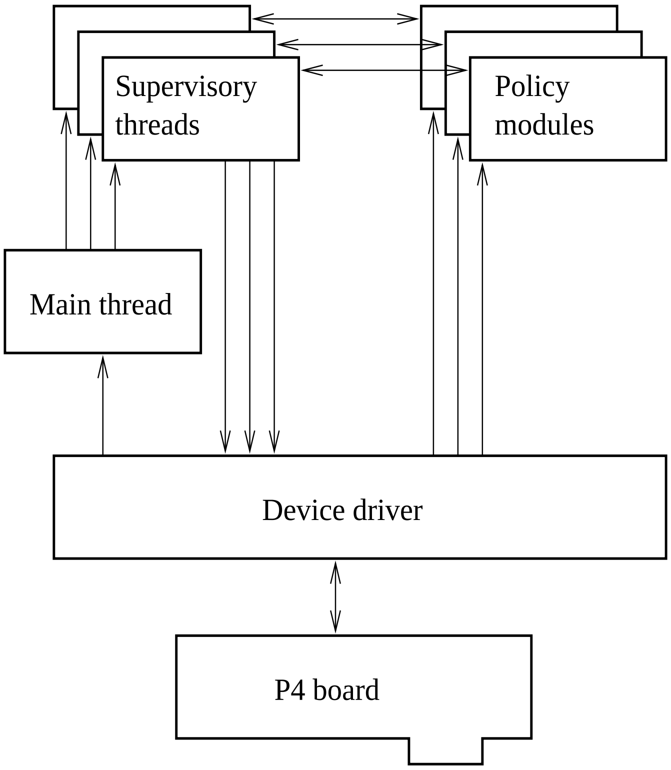

Figure 2: Software Architecture

to the appropriate supervisory threads. Each booster or debooster has the unique identification code and unique supervisory thread associated with it. The `DST` field is used to determine the supervisory thread which should receive the message. Messages with illegal `DST` field are discarded. Message with recognized `DST` field is stored in buffer for delivery to the appropriate supervisory thread, and the `cond_signal` is executed to wake up the thread.

Supervisory threads implemenet the signalling protocol. Each thread runs the same algorithm described in the next section. Policy modules are called by the supervisory threads and they interact directly with the device driver.

6 Signalling Protocol

For each booster or debooster there exists a separate supervisory thread executing the signalling protocol. The supervisory thread sleeps on `cond_timedwait` until woken up by the main thread or the occurrence of time-out.

After each wake-up, supervisory thread performs an action and changes state depending on its current state, reason it was woken up (message or time-out) and the content of the received message (if any).

If the message is not recognized in the given state, or if the message is illegal, it will be discarded, no action will be taken and the supervisory thread will not change its state.

When sending messages, supervisory thread interacts directly with the device driver through the `ioctl` system call.

Supervisory thread can be in one of the following states:

- `IDLE`: System is operating without the booster or debooster. Both booster and debooster can be in this state.

- `RUN_BOOST`: System is operating with the booster but it must keep sending `START` message because `START` has not been acknowledged yet (`STARTED` message has not been received yet).
- `RUN_DEBOOST`: System is operating with the debooster.
- `RUN_BOOST_SILENT`: System is operating with the booster and the `STARTED` message has been receive, so there is no need for sending signalling messages.
- `TERMINATING`: System is in transition from boosted to non-boosted mode. Thread keeps sending `ABORT` messages until it receives the acknowledgement.

Supervisory thread maintains a data structure which carries the information specific to that particular thread. The most important fields are its state, pointers to the policy modules for non-boosted and boosted mode, the identification of the target thread on the remote side (e.g. for the thread controlling the FEC encoder, a target thread is the the thread controlling the FEC decoder), associated FPGA configuration and the thread's role in the system (booster or debooster). Other fields are not essential for further discussion and thus they are not described in this document.

After initialization the supervisory thread starts in `IDLE` state and the associated booster or debooster is not loaded. On every time-out, a policy module runs. If there are no associated policy modules, the timer is stopped, and the thread remains in `IDLE` state. Depending on the policy module return value following actions are taken:

- Return value is 0: Timer is restarted and the thread remains in `IDLE` state.
- Return value is 1: Timer is restarted, `PREPARE` message is sent, and the free FPGA device is loaded, without switching it in the processing chain (`LOAD` operation)
- Return value is 2: Timer is restarted. If the thread is controlling the booster, the available FPGA device is loaded and switched in (`LOAD&SWITCH` operation), and `START` message is sent. If the thread is controlling the debooster, the available FPGA device is loaded but not switched in and the `BOOST` message is sent. Distinguishing the role of the thread in the system is necessary because only the booster side may initiate the transition to boosted mode, but any side can make the policy decision. The purpose of the `BOOST` message is to request from the booster to send the `START` message when ready.

In `IDLE` state, the following messages are recognized:

- `PREPARE`: Timer is restarted and the FPGA device is loaded but not switched in.
- `ABORT`: An `IDLE` message is sent in response. Timer is not restarted.
- `BOOST`: If the thread is controlling the booster, it loads the FPGA device, switches it in, restarts the timer, sends the `START` message and goes to `RUN_BOOST` state. If the thread is controlling the debooster, it loads the FPGA device without switching it in and restarts the timer, remaining in `IDLE` state.
- `START`: If the thread is controlling the booster, the message will be ignored because the debooster (the other side) is not allowed to send this message. If the thread is controlling the debooster, this message will mark the point at which deboosting should occur. The FPGA device will be loaded and switched in, and the `STARTED` message will be sent in response. Timer will be restarted and the the thread will go to `RUN_DEBOOST` state.

In `RUN_BOOST` state, two messages are recognized:

- ABORT: This message is received as the request from the remote side to remove the booster. Timer is restarted, the associated FPGA device is removed from the pipeline chain, and an IDLE message is sent in response. The thread goes to IDLE state.

- STARTED: Timer is restarted and the thread goes to RUN_BOOST_SILENT state, where it stops sending the START message.

If the timer expires in RUN_BOOST state, a START message will be sent again and the timer will be restarted. The state will not change.

Messages recognized in RUN_DEBOOST state are:

- ABORT: The same action is taken as in RUN_BOOST state. The thread goes to IDLE state.

- STARTED: The STARTED message is sent in response without restarting the timer. The state of the thread is not changed.

If the timer expires in RUN_DEBOOST state, a policy module associated with the boosted mode is run. If there is no policy module, the timer is stopped preventing further time-outs, and the state is unchanged. If the policy module returns 0 meaning that the booster should be turned off, the FPGA device is removed from the pipeline chain, the ABORT message is sent and the timer is restarted. The thread goes to TERMINATING state. If the policy module returns 1 or 2, meaning that the booster should be either turned on or kept in the current state (which is on) the timer is restarted and the state of the thread is not changed.

The thread is in the RUN_BOOST_SILENT state if the booster is switched in and there is no need to exchange the signalling messages so because the handshake has been achieved. Only the ABORT message is recognized and all other messages are ignored.

If the ABORT message is received, the timer is restarted, the FPGA device is removed from the pipeline chain and the IDLE message is sent. The state is changed to IDLE.

If the timer expires, the policy module associated with the boosted mode is run. If no policy module exists, the timer is stopped preventing further time-outs, and the state remains unchanged. The same action as in RUN_BOOST state is taken depending on the return value of the policy module.

In TERMINATING state, two messages are recognized:

- ABORT: Timer is restarted and IDLE message is sent in response. The thread goes to IDLE state. This case is possible if both sides have simultaneously decided to go to non-boosted mode.

- IDLE: Timer is restarted and the thread goes to IDLE state. Receiving the IDLE message indicates that the remote side has successfully switched to non-boosted mode and that there is no need to further send ABORT messages.

If the timer expires in TERMINATING state, the ABORT message is sent again, the timer is restarted and the state of the thread remains unchanged. In this way, the thread which has initiated the booster removal, will keep sending the ABORT message until it receives the IDLE message.

7 Limitations

The version of SPP4 described in this document does not support the device managing. That is, no sophisticated algorithm has been implemented to decide which FPGA device to overwrite when no blank device is available (i.e. replacement policy). Also there is no recovery process if there is no free device available (blank or configured). Deciding on the optimal set of functions which should be delegated in hardware and exploring the replacement policy is the objective of further research and it will be the integral part of the device driver and SPP4 in future.

Dynamic policy and mechanism module addition is not supported in this version and boosters known to the system are hard coded. This feature will be added in future versions.

References

[ATM94] ATM forum. *UNI 3.1 Specification*, 1994.

[BBD+96] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D Verworner. *Linux Kernel Internals*. Addison-Wesley, 1996.

[HS97] I. Hadžić and J. M. Smith. P4: A Platform for FPGA Implementation of Protocol Boosters. In *Field-programmable logic and applications: 7th International Workshop, FPL'97, Proceedings*, LNCS, 1304, pages 438–447. Springer, September 1997.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications*, 35(1):80–86, January 1997. *Earlier version MIT LCS TR #MIT/LCS/TM-557, 1996.*