1 An application in BPC: a Web-Server

We briefly describe our web-server case-study, dwelling in particular on some of the more advanced features of the BPC framework, such as timeouts, parametrized events, dynamic thread creation and customized event selection. We also try to convey to the reader a sense of the interaction between the behavioral code and native C^{++} code, and of the development process of a behavioral application.

1.1 TCP & HTTP

Our web-server application consists of two protocol stacks: one for *TCP* and one for *HTTP*.

The *Transmission Control Protocol* (*TCP*) is a widespread connection-oriented protocol, mainly used for the transfer of data across the internet. The principle feature of TCP is its reliability: it guarantees that data arrives without errors and in the order in which it was sent. To accomplish this, the end parties acknowledge the reception of each TCP segment using a scheme of agreed-upon sequence numbers; segments that are lost or arrive corrupt are not acknowledged, and are then retransmitted. As TCP entails redundancy in the bits being sent, it contains flow and congestion control features, meant to prevent overwhelming an endpoint or clogging the network.

The *Hypertext Transfer Protocol* (*HTTP*) is a protocol typically used by web-servers. The client (the browser) submits HTTP requests to the server, which then performs local actions on behalf of the client and sends responses; in particular, the server might retrieve webpages, send error or redirection messages, or run scripts locally and send the their output to the client.

1.2 The Implementation's Layout

Our application consists of two separate sets of threads, one for the TCP layer and one for the HTTP layer. The two layers interact with each other via behavioral events — that is, each has a thread constantly waiting for certain events generated by the other. Each layer also has one additional source of input: the TCP layer reads TCP segments off a "raw socket", and the HTTP layer reads files from a given directory. These additional inputs are obtained by threads containing non-trivial native C^{++} code, and are then translated into behavioral events in order to be passed to other threads.

Internally, each layer is designed using a *dispatcher* architecture: a dispatcher thread handles each incoming segment, classifies it according to its attributes, and then passes it to specific *handler* threads via behavioral events. These handler threads can then request additional events in order to issue a reply and/or update other threads of the contents of the segment. Handlers typically perform local computation using native C^{++} code — e.g., calculating TCP checksums or reading files from the directory. Incoming TCP segments containing HTTP requests are passed between the layers, and the same happens to HTTP replies on their way to the client.

To exemplify the server's operation we describe in more detail the handling of TCP connection establishment requests (SYN segments). Initially, the *RawSocketReceiver* sensor thread reads the incoming segment from the socket. When it is received, the thread requests a tcpSegmentReceived event — with the segment as its parameter — in order to pass the segment to the *TcpDispatcher* thread.

The *TcpDispatcher* uses native C^{++} code to classify incoming TCP segments according to their attributes, and requests additional events accordingly. SYN requests, for example, are identified by reading the TCP header of the segment and checking whether the SYN flag is set. If so, the thread requests a tcpSynRequest event in order to notify *TcpSynHandler* — the specific handler for SYN requests.

The *TcpSynHandler* thread responds to each request by generating a tcpOutgoingSegment event, with a SYN-ACK segment as its parameter. Finally, this event then gets translated into a tcpSendSegment event — handled by the *RawSocketSender* sender thread, which actually sends the SYN-ACK segment to the client.

We note that in order to construct the SYN-ACK segment, the *TcpSynHandler* thread must first acquire a fresh sequence number. This is performed by sending a request to, and receiving a response from, the *SequenceNumberAllocator* thread — the thread in charge of managing the sequence numbers of every TCP connection. *SequenceNumberAllocator* may handle simultaneous requests for sequence numbers (for the same connection) from multiple threads, and here the BP event selection mechanism guarantees that each outgoing segment has a fresh sequence number: *SequenceNumberAllocator* handles requests (represented by events) sequentially, and thus race conditions are avoided.

Apart from dispatcher and handler threads, additional "standalone" threads exist in the system: for instance, the requirement that TCP segments be sent only on active connections is enforced by the *TcpEnsureActiveConnection* thread. This thread uses blocking to ensure that a tcpSynRequest is triggered before other TCP events — such as those signalling PUSH or ACK segments — are triggered. Likewise, once a FIN segment is triggered, the thread blocks any additional TCP events for that connection.

Segment Reordering TCP segments that contain data for the HTTP layer are not guaranteed to arrive in the order in which they were sent. Hence, the TCP layer needs to reorder them before passing them on.

During data transfer, TCP segments with data for the HTTP layer cause the triggering of dataToHttp events. Each of these events carries the received segment's sequence number as a parameter. The TCP stack knows the expected sequence number of the next data segment: the initial sequence number is stated by the client at the time of connection establishment, and is subsequently incremented for each segment. Whenever an incoming sequence number is greater than the one expected, the stack realizes that a segment is missing; and when this segment later arrives, reordering takes place. Pseudocode for the *SegmentSorter* thread, which is in charge of this reordering, appears in Fig. 1.

```
void entryPoint() {        // SegmentSorter thread
   while( true ) {
      Vector<Event> requested, waited, blocked;
      waited.append( tcpSynRequest );
      waited.append( dataToHttp );
      bSync( requested, waited, blocked, NO_TIMEOUT );

      e = lastEvent();
      if ( e.type() == tcpSynRequest )
         storeSequenceNumber( e.sequenceNumber() );
      else
      if ( e.sequenceNumber() != expectedNumber() )
         storeData( e.data() );
      else sendReorderedSegments( e.data() );
}};
```

Fig. 1. The *SegmentSorter* thread waits for tcpSynRequest and dataToHttp events. When a tcpSynRequest event occurs, the thread extracts the sequence number for later use. When an actual data segment is received, its sequence number is compared to the expected number. If it does not match, the segment is stored. If it does match, the segment is passed on to the HTTP layer, along with any consecutive segments previously stored, and the expected sequence number is updated.

Segment Retransmission When sending out a segment, the TCP stack must wait for an acknowledgment message — and if one does not arrive, the segment needs to be resent. There exist several sophisticated retransmission policies, aimed at reducing traffic congestion, which have adjustable retransmission periods. For our case-study we opted for the simplest scheme — retransmission after a fixed waiting period. Implementing additional schemes is left for future work.

In our implementation, segments leaving the TCP stack on their way to be sent to the client always pass as tcpOutgoingSegment events. Our retransmission mechanism waits for these events and stores the outgoing segments. Then, if they are not acknowledged withing a fixed period of time, it retransmits them. Pseudocode appears in Fig. 2[1].

Customized Event Selection During development, we occasionally found customizing the event selection strategy a straightforward method in order to enforce certain requirements. For example, in one case we wanted to ensure that all outgoing segments finish sending prior to sending the segment indicating the connection being closed (a FIN segment) — a property that was not trivially

[1] The depicted solution spawns a thread dynamically for each outgoing segment, which incurs overhead. In practice, we found that it was more efficient to spawn one *Retransmitter* thread per connection, and have it handle all of that connection's segments. Nevertheless, we feel Fig. 2 better illustrates the principles of dynamic thread creation.

```
class Retransmitter : public BThread {
   void entryPoint() {
      Vector<Event> requested, waited, blocked;
      waited.append( tcpOutgoingSegment );

      while ( true ) {
         bSync( requested, waited, blocked, NO_TIMEOUT );
         new PeriodicSender( lastEvent() );
}}};

class PeriodicSender : public BThread {
   PeriodicSender( Event tcpOutgoingSegment ) {
      storedSegment = tcpOutgoingSegment;
   }

   void entryPoint() {
      bool done = false;
      while ( !done ) {
         Vector<Event> requested, waited, blocked;
         waited.append( ackForStoredSegment() );

         bSync( requested, waited, blocked, 2 );
         if ( timeoutOnlastSync() ) {
            waited.clear();
            requested.append(
                    tcpSendSegment( storedSegment ) );
            bSync( requested, waited, blocked, NO_TIMEOUT );
         }
         else done = true;
}}};
```

Fig. 2. Pseudocode for the segment retransmission mechanism. The *Retransmitter* thread waits-for tcpOutgoingSegment events — events that indicate a TCP segment about to be sent — and for each such event it spawns an instance of the *PeriodicSender* thread. The *PeriodicSender* instance receives through its constructor the segment that it is supposed to monitor. It then waits for an acknowledgment of that segment for 2 seconds. If an acknowledgment message fails to arrive, the thread retransmits the segment, and the process repeats. When an acknowledgment is received, the thread terminates. Note that the `ackForStoredSegment` method (code omitted) is a *predicate* — it evaluates to true only for tcpAckReceived events with the proper acknowledgment information.

upheld by the TCP stack. Another example was giving priority to starving connections, namely connections whose events have not been triggered in a while, in order to avoid retransmission of segments and the congestion incurred by it.

Pseudocode for a customized event selection function that addresses these two issues appears in Fig. 3.

```
Event choose( Vector<Event> enabledEvents ) {
   Vector<Event> candidates =
      eventsOfStarvedConnection( enabledEvents );

   if ( candidates.contains( sendTcpFin ) &&
       candidates.containsOtherThan( sendTcpFin ) )
      return candidates.otherThan( sendTcpFin );
   else return candidates[0];
};
```

Fig. 3. Pseudocode for the customized event selection strategy. At every synchronization point, this function is invoked with the set of enabled events, of which it must select one for triggering. Information from previous iterations may be stored. Our specific implementation gives precedence to previously "starved" connections: that is, it favors the connection that has waited the longest for an event to be triggered. This part is abstracted away in the method `eventsOfStarvedConnection`. Once a connection is selected, its associated events are the `candidates` for triggering; among these, we prefer events that are not `sendTcpFin`, so that pending data transmission requests are addressed before the connection is closed. Otherwise, an arbitrary event is selected.