♦ Measuring and Simulating Cellular Switching System IP Traffic

Kenneth W. Del Signore

The increasing popularity of smartphones and other data-enabled cellular devices has led to a rapid expansion of cellular data networks worldwide. These networks are constantly being improved to provide new functionality, improved capacity, and better reliability. Optimization, testing, and troubleshooting of these systems are all important areas of concern within the cellular network industry. Large amounts of engineering resources are devoted to these topics, with the result that analysis techniques and best practices are constantly being improved. This paper details recent work regarding the analysis of cellular data device behavior in a live system, and the design of a high performance load generator to simulate this behavior for use in system testing and debugging. Internet Protocol (IP) packet capture data is analyzed using a simple graph-based analysis technique that provides a detailed characterization of device behavior. This characterization is then used in the design of a second generation high performance load generator. © 2014 Alcatel-Lucent.

Introduction

Data enabled cellular switching systems provide a critical function in modern society, namely to allow cellular data devices including smartphones, tablets, data cards, and machine-to-machine applications to access the Internet. This process is shown schematically in **Figure 1**. The data device transitions from a dormant to an active state by executing a "connection" to the switching system through one or more cell towers. The device then exchanges data over the Internet and then transitions back to the dormant state when the data transfer is finished.

Switching systems such as shown in Figure 1 are highly engineered towards the goal of providing fault-free service. Service interruptions can have many adverse effects, for instance public safety can be affected by not being able to access the system in an emergency, or more commonly, users are inconvenienced to various degrees by loss of service. Service interruptions can also adversely affect the profitability of the system operator, such as through lost revenue due to network downtime or customer churn due to poor network service; this in turn can lead to loss of future equipment sales for the manufacturer of the switching system. For these reasons and others, considerable resources are expended to improve the reliability of these switching systems.

In order to minimize faults, cellular switching systems are subjected to extensive software and hardware testing before each new version of system software or hardware is released. This pre-release testing catches many faults, however the systems still experience a large number of faults post-release in live field environments. This is evidenced by the large number of engineers assigned to "field support" duties within the cellular network industry. It is therefore a constant goal within the industry to improve pre-release software and hardware testing.

A load generator, which is a dedicated server with purpose-built software, is a critical tool for pre-release testing. It connects to the switching system and simulates the messaging load that a large number of data devices would generate. The data devices interact with the switching system through messaging scenarios consisting of sequences of messages. There are multiple possible messaging scenarios that the data devices can generate. The specific scenarios generated and their respective frequency of occurrence is referred to as the "call model." There are three main scenarios in the call model, commonly referred to as registration, connections, and mobility management. Registration is the process that cellular data devices use to register for service with the switching system. Following registration, the devices execute connections to the switching system to enter the connected (or active) state in which they can exchange data over the Internet; when not exchanging data, the devices revert to an unconnected (or

Panel 1. Abbreviations, Acronyms, and Terms

3G—Third generation
4G—Fourth generation
EV-DO—Evolution Data Optimized
I/O—Input/output
IP—Internet Protocol
KPI—Key performance indicator
LTE—Long Term Evolution
PDF—Probability distribution function
RF—Radio frequency
WNG—Wireless Network Guardian

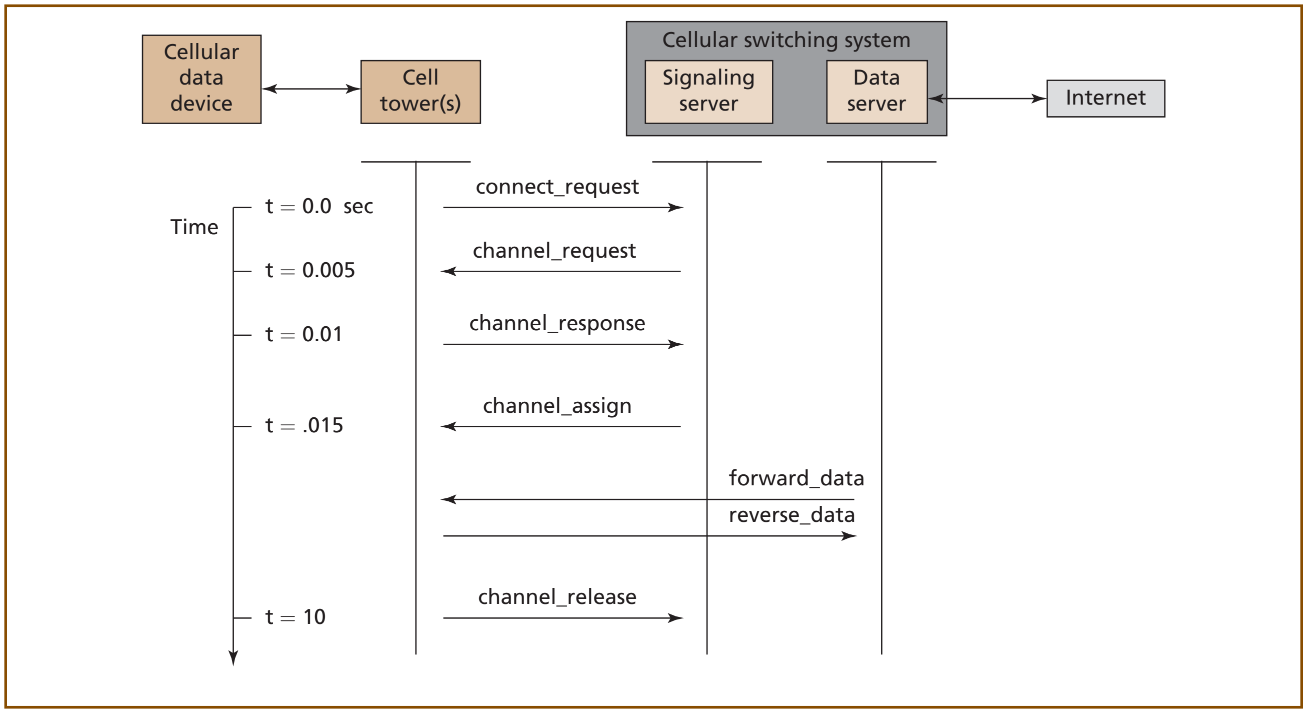

Figure 1.
A generic diagram of a "data connection" in a cellular data network.

dormant) state to conserve system resources. Mobility management refers to the operations needed to allow the devices to physically move throughout the service area (both in the connected and unconnected states).

In practice, it has proven challenging to design load generators that emulate the complexity seen in a live field system call model. Here *complexity* refers to variations of the call model around the normal scenarios. For instance, a device may initiate a connection and then some error may occur that causes the attempted connection to fail. Another example seen in the field is when some combination of the three main call model scenarios occur simultaneously, often resulting in an error. Faults in live field switching systems are often tracked to low probability error scenarios generated by a data device. By examining messaging data from live field systems, we observed that there are many more unique messaging scenarios present than are generated by a typical load generator used for pre-release testing. The lack of complexity in the load generator call model in effect means that the switching system is not tested in the same environment that it will experience at a live field site.

There are several factors that contribute to this lack of complexity. The main factor is that the exact details of the field call model are difficult to measure in a live switching system. Without knowing the exact messaging scenarios, the load generator cannot be made to simulate them. This measurement difficulty is due to the fact that scarce system resources cannot be dedicated to measuring these details. Instead, the call model is defined using high level counters, referred to as *key performance indicators* (KPIs), maintained by the switching system. KPIs are generally reported as hourly averages aggregated over all devices. These counters report measurements such as the number of unique data devices registered for service, the total number of connections made by all devices, the total time spent in the connected state, and the total data volume to and from the data devices. While KPIs are adequate to monitor details of high-level system performance, many details about the complexity of the call model are lost in the aggregation process.

The problem of measuring call model details in the field can be overcome by taking an external measurement of the messages exchanged between the data devices and the switching system. These messages are encoded in Internet Protocol (IP) packets which pass through a switch situated between the cell towers and switching system. These packets can be captured at the switch and processed offline to reconstruct the exact messaging scenarios that occurred during the time of the data capture. Further, the packet capture can be done remotely with no degradation to overall system performance.

A secondary factor for the lack of complexity found in most load generator call models is that load generators are usually quickly designed and pressed into service when a new switching system is first developed. Important details regarding the field call model and the load generator design required to simulate it are often not known at this stage; consequently load generator software is not designed in such a way that these details, once uncovered, can be easily implemented. This effect is well-known in software development projects [2], namely that the second generation of a software system will benefit greatly from the lessons learned in the first generation.

A third factor that contributes to load generator design problems is that individual load generator projects are usually done by different teams of engineers within an institution, separated in time by many years. As a result, knowledge gained on one project is not readily transferred to the next. Many simple and important details, such as the generation of connection attempts with a Poisson distribution, can be skipped in a newly designed load generator.

The paper addresses all three of these factors. The load generator design presented in this paper was developed as a result of approximately ten years of field experience with the Alcatel-Lucent Evolution Data Optimized (EV-DO) switching system. It represents a second generation design, based on this experience. Packet capture data from a live field system is used to measure and define the call model scenarios and the software design of the load generator is such that these scenarios can be easily implemented and expanded upon when desired.

The paper is organized as follows. We begin with a discussion of related work in IP packet analysis and load generators. Next, a generic connection scenario is defined which introduces the components of a cellular switching system and describes the IP packet-based messaging that occurs during each normal (or "sunny day") connection. Following that, we describe how IP packet data is collected from messages in a live switching system. The messages in this dataset are then aggregated into a graph structure which exposes various characteristics of the underlying call model. The main sunny day connection scenario and two error scenarios are then explored using the graph structure. The information gained about these scenarios is then applied to the design of a high performance load generator. The paper concludes with a discussion of some considerations involved in designing test strategies for cellular switching systems.

Related Work

There are a large number of open source and commercial software utilities designed for IP packet traffic analysis on Ethernet networks. These utilities usually decode each packet in real time to collect and aggregate network related statistics, then discard the packet after the desired information is extracted. The most common purpose of these utilities is to inform network administrators about the nature of the traffic within their network. The IP traffic analysis in this paper is somewhat different from this approach in that the packet data is stored, and then iteratively (re)analyzed to explore long sequences of packets that come from each unique device.

One commonly used open source utility that works with stored IP packet data is Wireshark* [7]. This utility reads-in packet capture data in the raw format it is transmitted in and then decodes each Ethernet frame and the encoded protocol layers within. Many open standard protocols are supported and the utility can be extended to support proprietary protocols. The data used in this paper is first pre-processed with an Alcatel-Lucent proprietary extension of Wireshark as described below.

Alcatel-Lucent has developed multiple commercial products based on IP packet capture data. Two such products are Celnet Xplorer [4] and Wireless Network Guardian (WNG) [1]. Both products use dedicated hardware placed within the IP network serving a switching system to continuously capture IP packet traffic. Information is extracted from each packet and aggregated in a second server for report generation. WNG, which provides many important details of real-time network status, is actively being used by cellular network operators in North America. The analysis presented in this paper makes a field call model measurement based on a one time collection of packet capture data, which is done remotely with no external hardware. This technique is thus more suitable for specific analyses, such as solving specific field problems, but could be applicable to WNG for some uses, such as long term monitoring of call models.

A recent paper on load generators [6] discusses many of the issues involved in designing a state-of-the-art load generator, and that work is partially based on experience with the same first generation load generator as this paper. However, in [6], the authors focus on simulating the data that is exchanged between the Internet and the data device when in the active state, whereas this paper will focus on the signaling messages used to transition between the active and dormant state. They closely follow the work of [5], which proposes to represent this process as a series of Markov states with each state having a predefined data rate and transition probability to the next state. The packet analysis technique used in this paper can be used to define and measure such states and transition probabilities.

A Generic Connection

Figure 1 depicts a generic connection scenario that will be used throughout this paper. When a cellular data device needs to transmit or receive data, such a connection procedure is executed. The scenario allows the data device to transition from a dormant to a connected state. In the connected state, the device can access the Internet. After some period of inactivity, the device transitions back to a dormant state to conserve resources. The scenario proceeds by the transfer of messages between the signaling and data servers and the cellular data device via the cell tower(s) in radio frequency (RF) contact with the device. If no errors occur, it is referred to as a sunny day connection.

A data device exchanges messages (denoted with horizontal arrows) with the switching system to transition from a dormant to an active state. The data device first sends a connect_request message to the signaling server to request a connection. The signaling server then sends a channel_request to the cell site(s) in RF contact to request RF channel(s). The cell site(s) each respond with a channel_response message. The signaling server then sends a channel_assign message to the data device, instructing it to tune to the RF channel and enter the connected state. The device can then send IP packets through the data server to the Internet. When the device is finished exchanging data it sends a channel_release message and returns to the dormant state.

Data Collection

The packet capture data used for this paper was collected from a live EV-DO field switching system. This system was serving approximately 100 cell towers and 150,000 unique EV-DO data devices. These devices were generating approximately 1.4 million connection attempts per hour at the time of the data collection. Data was collected for 25 seconds and there were 9.8K connection attempts during this interval. The size of the collected data file is 1 GB.

The packet capture data is processed via an Alcatel-Lucent proprietary extension of Wireshark. This produces a verbose decoding of each Ethernet frame as shown in **Figure 2**. Each frame contains one or more encoded messages which are also verbosely decoded.

Each message type as well as other desired data is stripped from the large stream of verbose decoded frames and stored as a file of smaller message records. These records are then sorted by a unique data device ID present in each message. This produces a list of messages that each data device exchanged with the switching system during the time period of the packet capture.

For the purposes of this paper, the analysis will focus only on messages to and from the signaling server in Figure 1. This will suffice to describe the analysis method and how the resultant observed device behavior is applied to a simulator.

Graph Based Analysis Algorithm

The following algorithm was developed by Alcatel-Lucent, is used internally, and has been very useful in analyzing certain types of data generated in cellular switching systems. In the present analysis we wish to extract call model information from the raw IP packet capture data. The graph algorithm will permit this extraction with relative ease.

The algorithm works by building a graph (or tree) from the sequences of messages that each data device generates. The printout of the resultant graph displays useful information about the underlying call model. To further refine the displayed information, special graph nodes are introduced that perform simple functions on the graph. These function-nodes will be used to modify the graph (to further refine the displayed information) and to extract other useful information.

The process proceeds iteratively: the graph is built and printed out, function nodes are inserted, and a new graph is rebuilt and printed. The cycle continues until some desired information is obtained. This process will be used to investigate the three most commonly occurring call model scenarios in the IP packet capture data.

Two notable aspects of the algorithm are that it permits a detailed investigation of the data with relative ease and that this is done with a minimal amount of code. The use of code to extract useful information from large datasets is fundamental to virtually all areas of modern research, and any given problem can typically be solved in multiple ways. Code-based analyses typically proceed iteratively, with code written at each step to further refine the preceding analysis result. Such techniques can be a manual process for the investigator and can be prone to bugs. Success

```
Frame 35669 (145 bytes on wire, 145 bytes captured)
    Arrival Time: Jan 4, 2013 17:18:30.646692000
    ...
    [Protocols in frame: eth:ip:tcp:evdo_rmi:evdo_slp]
Ethernet II, Src: Vmware_03:00:4f (00:50:56:03:00:4f), Dst: All-HSRP
    -routers_19 (00:00:0c:07:ac:19)
    ...
Internet Protocol, Src: 10.104.25.6 (10.104.25.6), Dst: 172.26.3.6
    (172.26.3.6)
    ...
    Source: 10.104.25.6 (10.104.25.6)
    Destination: 172.26.3.6 (172.26.3.6)
Transmission Control Protocol, Src Port: 33825 (33825), Dst Port:
   x11 (6000), Seq: 8293, Ack: 4999, Len: 79
    ...
Lucent EV-DO RMI (SBEVM)AccessChannelPacketInd
    ...
    Access Channel Packet (24 bytes)
        MAC Header UATI 0x15200881
           ...
        MAC Layer Payload (Format B)
            Connection Header Length: 7
            Connection Layer Payload
               ...
               SNP Message (RouteUpdate)
                   Route Update Protocol
                       RouteUpdateMsgType: RouteUpdate (0)
                       MessageSequence: 1
                       10000010 0....... = ReferencePilotPN: 260
                       .000101. ........ = ReferencePilotStrength:
                          5
                       .......1 ........ = ReferenceKeep: 1
                       0000.... ........ = NumPilots: 0
            Connection Header Length: 5
            Connection Layer Payload
               ...
               SNP Message (ConnectionRequest)
                   0... .... = InConfigurationProtocol: In Use (0)
                   .000 1100 = Type: Idle State Protocol (12)
                   Idle State Protocol
                       DefaultIdleMessageType: ConnectionRequest
                          (1)
                       TransactionID: 0
                       0000 .... = RequestReason: Access Terminal
                          Initiated (0)
```

†Registered trademark of the Wireshark Foundation, Inc.
IP—Internet Protocol
EV-DO—Evolution Data Optimized
RMI—Remote method invocation
TCP— Transmission Control Protocol

Figure 2.
A Wireshark† verbose decode of an IP frame sent between a cell tower and a switching system. Wireshark decodes the Ethernet, IP, and TCP layers. The Alcatel-Lucent proprietary extension to Wireshark decodes the proprietary EVDO_RMI layer. Approximately half of the decoded lines are omitted for brevity. This frame is an example of a connection request message sent by the data device.

of the final result is easier to achieve if the investigator's work at each step is simplified. In the present analysis, the graph algorithm greatly simplifies the task of extracting detailed call model information from the data. This is due to a combination of the compact nature of the graph printout and the iterative use of a small number of function nodes to refine the graph.

The remainder of this section is organized as follows. We first provide a simple example of a graph. The code required to build the graph is shown and an example of the graph construction process is further detailed. Following this, the main() routine used in the analysis is described and the specific code to build a graph from the message data is shown. The initial graph is then modified with a function node to prune unneeded information (messages) in order to better aggregate the information displayed by the graph. The relatively simple code (three lines) to implement this function node is given. Two additional function nodes are described and then these three function nodes are used to investigate and detail the main sunny day connection scenario and the two most common error leg scenarios.

As an example of a simple graph, consider the following collection of sequences:

```
[A,B,C]
[A,B,D]
[A,E,F]
[G,B,C]
[G,B,D]
```

Here each letter can be thought of as corresponding to a unique message type while each sequence corresponds to a unique data device.

For the design of a load generator, the first question to be asked is: if message "X" occurs, what are the next possible messages that can occur, and with what probabilities? This question then repeats iteratively for the next message(s) that will occur, and continues to repeat until a complete messaging sequence is executed. This question can be answered exactly by aggregating the sequences above into a graph. The graph built from the above sequences would be:

```
3:A
        2:B
                1:C
                1:D
        1:E
                1:F
2:G
        2:B
                1:C
                1:D
```

The equivalent graph representation drawn with nodes and edges is shown in **Figure 3**. In the above representation each line represents a node of the graph. Branches of a node are indicated by the subsequent lines with increased tab level. The first line, "3:A," indicates that the symbol "A" occurred as the first symbol in three sequences. The second line, " 2:B", indicates that the symbol "B" was the second symbol following "A" in two of the sequences. The third and following lines all follow the same pattern.

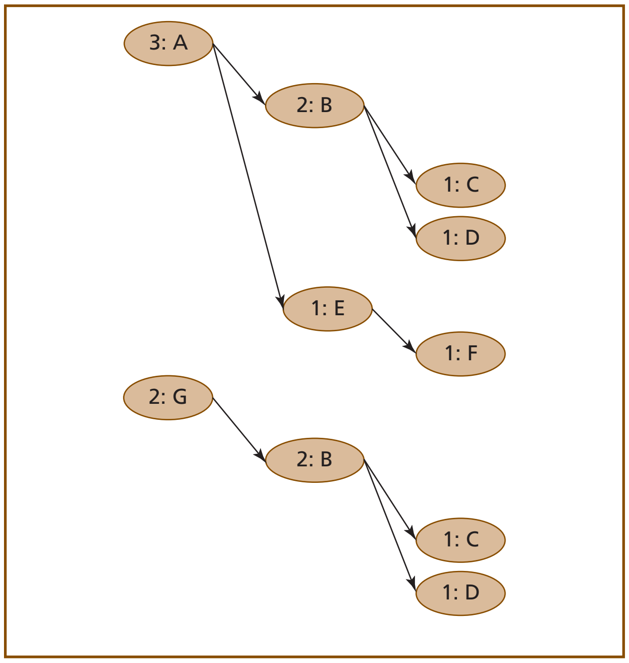

Figure 3.
An example graph used in the text that is drawn with the classic nodes and edges style. As used in the text, this is a directed graph that can be alternatively referred to as a rooted tree.

The message sequences in live systems tend to have a few sequences that occur most often (sunny day scenarios), with other sequences that occur less often (rainy day scenarios). As an example, consider the previously used five unique sequences now modified with the following occurrence counts:

```
[A,B,C] × 1000
[A,B,D] ×  100
[A,E,F] ×  200
[G,B,C] ×    1
[G,B,D] ×    5
```

This data would aggregate to the same graph structure as before, but with the counts modified as follows:

```
1300:A
      1100:B
            1000:C
             100:D
      200:E
             200:F
6:G
        6:B
               1:C
               5:D
```

The graph representation provides a useful representation of the sequence data. For instance, branching ratios can be inspected from the graph, and also low probability sequences are exposed that may otherwise be obscured by the large dataset. Note also that branching ratios are captured with context, for instance the probabilities to transition B $\rightarrow$ C,D depends on whether B is in the branch of the A or G root node.

The implementation of the graph structure is easily done as a recursive class, as shown in **Figure 4**. Each node of the graph is an instance of this class. Three member variables are needed: a string to uniquely identify a node (as in A, B, C …), an integer counter, and a list of branch nodes. The class has two recursive functions, one to input data and one to output data, add_sequence() and print_out() respectively. The functions are shown in Figure 4b and Figure 4c as pseudo code.

As an example of the recursive construction of a graph, consider the sequence of graph objects denoted by [A, B, C, -stop]. Here the "-stop" object is a flag to terminate the recursive add_sequence() calls. The graph will be built under a single node declared as "root." **Table I** shows the structure of the graph at each of the recursive add_sequence() calls. As a further example, assume the sequence [A, B, D, -stop] is added to the same graph. **Table II** shows the updated structure of the graph at each add_sequence () call.

For the call model analysis, the sequences of messages that each data device generated are used to build a single graph. The focus will be on the connection scenario that begins with the connect_request message of Figure 1. The devices can be in any state when the data capture begins, therefore the connect_request message may not be the first message captured. Also the devices may make multiple connect_request messages during the capture. This can complicate the interpretation of the resultant graph. To overcome this, each message sequence is added to the graph by treating each message of the sequence as the starting point for a new sequence. This has the effect that all occurrences of a message type will be aggregated into a top level graph object. As an example, if the message sequence generated by a data device is [A, B, C, A, B, D], the graph would be built with the following sequences:

```
[A,B,C,A,B,D]
  [B,C,A,B,D]
    [C,A,B,D]
      [A,B,D]
        [B,D]
          [D]
```

If the "A" symbol represents the connect_request message, then it can be seen that two of the above sequences start with "A," which allows this message to be studied in detail by focusing on the portion of the graph with "A" as its root. The pseudo code for this process is shown in **Figure 5**.

The print_out() statements in Figure 5 will print out the graph at successive levels of depth. This form of output is useful for analysis. By searching this output for unique "count:message" pairs, one can examine a specific message sequence at successive levels of depth.

```
(a) class Graph{

      string        symbol
      int           count
      list<Graph>   branches

      add_sequence( list<Graph>::iterator s )
      print_out   ( int depth, int tabs     )

    }

(b) //------------------------------------------------
    //
    Graph::add_sequence( list<Graph>::iterator s ){

      if (s->symbol==”-stop”) return;

      loop ( branches.begin to end){
        - find object with matching symbol as s->symbol
        - create and insert new object if needed
      }

      Graph* b= matching Graph object in branches list

      b->count++
      b->add_sequence( ++s )

    }

(c) //------------------------------------------------
    //
    Graph::print_out( int depth, int tabs ) {

      if (depth <= 0) return;

      loop (i = 0 to tabs) { cout<<”\t” }

      cout<< count<<”:”<<symbol<<endl

      loop (b = branches.begin to end){
          b->print_out( --depth, ++tabs)
      }

    }
```

Figure 4.
The Graph class used to implement the graph structure.

The 25 seconds of field packet capture data is processed as described. **Figure 6** shows the first level printout of the message graph, which provides a list of message types sorted by frequency of occurrence. The counts are skewed on some of the messages due to the capture settings. In particular, the fwd_data and rev_data packets are scaled by a factor of 1/24. The page messages are sent from the signaling server to the cell towers to locate the data device prior to executing the connection scenario. The fwd_data and rev_data messages contain data packets exchanged between the data device and the Internet.

There are two features of the data in Figure 6 that will bear in the subsequent analysis. There are

Table I. Example recursive graph construction from the sequence [A, B, C, -stop].

Step	Function call	Graph structure
1	root.add_sequence(A)	0:root
2		0:root
	A.add_sequence(B)	1:A
3		0:root
		1:A
	B.add_sequence(C)	1:B
4		0:root
		1:A
		1:B
	C.add_sequence(-stop)	1:C

Table II. Example of a second sequence [A, B, D, -stop] added to the graph of Table I.

Step	Function call	Graph structure
1	root.add_sequence(A)	0:root
		1:A
		1:B
		1:C
2		0:root
	A.add_sequence(B)	2:A
		1:B
		1:C
3		0:root
		2:A
	B.add_sequence(D)	2:B
		1:C
4		0:root
		2:A
		2:B
		1:C
	D.add_sequence(-stop)	1:D

messages appended with an "_i". These are generated during an internal scenario specific to the Alcatel-Lucent system. They occur when two signaling servers are used in a connection scenario. These internal messages are not going to be examined in this analysis; however, they are left in to demonstrate a graph pruning technique that will be used.

It can also be seen that for every connect_request message—line 9814 (the message count will be used to refer to specific lines in the figures)—there are more of the subsequent sunny day connection scenario messages (lines 19002, 26086, and 26148). These counts are approximately proportionally correct. The difference is due to the data device using multiple RF paths in a connection. These messages are sent to each individual tower face that has contact with the data device (as reported in the initial connect_request message). There can be one to five of these RF paths in a single connection. A second, recursive, pruning technique will be used to collapse

```
main(){

  Graph message_graph("message_graph")

  loop over all devices{

     list<Graph> messages = sequential messages to/from device

     loop (m = messages.begin to end){

        message_graph.add_sequence( m )

     }

  }

  message_graph.print_out( 1, 0 )
  message_graph.print_out( 2, 0 )
  ...
  message_graph.print_out( n, 0 )

}
```

Figure 5.
The graph is built under a single root node declared in main() as "message_graph."

these multiple messages into a single message in the graph.

Focusing on line 9814 of Figure 6, the first and second level printouts of the connect_request portion of the graph are shown in **Figure 7**. In Figure 7, lines 6100 through 1 show which messages followed the connect_request message along with their associated counts**.** In the case of line 6100, we wish to prune these internal messages and re-aggregate the graph. To accomplish this, a technique is introduced that places a flag node (a Graph object with its symbol set to some predefined flag) in the message graph prior to re-aggregating the graph. This flag will trigger code to run in the add_sequence() routine that will do the pruning. Additional flags will be defined as needed by the analysis. This technique will allow for easy implementation of analysis code that runs deep within the graph at points specified by the investigator as needed. The technique is initially more complicated than for instance simply filtering out the internal messages; however the advantages become apparent when used in complex scenarios. The flag node technique allows the investigator to quickly explore the data given the graph printout. Also, once the specific flag code is implemented and debugged, it can be reused without having to introduce more code (and possible bugs) into the analysis.

In the case of pruning a message from an input sequence, a flag node denoted by the symbol "-skip" is defined which causes add_sequence() to skip the node that precedes it. As an example, if the sequence [A, B, -skip] is placed in a graph, then all subsequent sequences [A, B, C, D] would be graphed as [A, C, D]. The use of the node and its implementation are shown in **Figure 8**. In Figure 8a, the placement of the node is shown in an existing graph. The node is inserted into the graph prior to re-aggregation by the bold code in Figure 8c. When the node is encountered by the add_sequence() routine, the bold code in Figure 8b is run which has the effect that the preceding node is pruned from the input sequence. The resultant graph is shown in **Figure 9,** in part (a) of the figure.

A similar result could be trivially obtained by filtering out all occurrences of a message, however the

```
0:message_graph
 996873:page

 223395:page_i

  30655:channel_request_i

  30158:channel_release_i

  26267:channel_release

  26148:channel_request

  26086:channel_response

  23669:channel_response_i

  21742:channel_assign_i

  19002:channel_assign

  10966:connect_request_i

   9814:connect_request

   5247:rev_data

   5122:fwd_data

   ...
```

Figure 6.
Messages that occurred during the 25 second packet capture sorted by message count. The page messages are sent from the signaling server to the cell towers to locate the data device prior to executing the connection scenario. The fwd_data and rev_data messages contain data packets exchanged between the data device and the Internet.

```
9814:connect_request

9814:connect_request
      6100:connect_request_i
      3314:channel_request
       297:channel_request_i
        48:channel_assign
        23:channel_release
        22:session_close
         8:channel_release_i
         1:connect_request
         1:session_request
```

Figure 7.
The connect_request message and the next messages that occurred. The "session_*" messages are from registration scenarios.

programmatic graph node technique allows for selective pruning at a specific location in a graph, for instance this could be done if one wanted to prune only the first occurrence of "B" in the sequence [A, B, C, B, D].

In the remainder of this paper, two additional programmatic node types will be used. These nodes will allow the large, relatively complex, message capture dataset to be explored and call model information to be extracted. The main sunny day connection scenario and the first two most frequent error legs will be examined.

A second programmatic flag "-skip_r," is introduced that will be useful in the remaining analysis. It has the effect in add_sequence() that any occurrence of an associated symbol will be pruned for the duration of the current recursive add_sequence() calls. For instance if the sequence [A, B, C, B, D, B, E] was aggregated into a Graph that was pre-loaded with the code sequence: [A, B, -skip_r, B], the resultant pruned sequence would be graphed as [A, B, C, D, E]. Here the sequence [-skip_r, B] causes add_sequence() to skip any subsequent occurrence of the "B" symbol in the input sequence. The code for this function node is implemented in a fashion similar to the "-skip" node in Figure 8b.

The -skip_r function is needed to collapse multiple combinatoric message sequences into a single graph branch. In the present data, the second and third messages in the sunny day connection scenario can repeat up to five times based on how many tower faces the device is in RF contact with. These messages can occur in different orders such as

```
[A,B,C,B,C,D]
[A,B,B,C,C,D]
```

For the analysis presented here, the -skip_r function will be used to reduce both of these messaging combinations into:

```
[A,B,C,D]
```

```
(a)    9814:connect_request
             6100:connect_request_i
                                  -skip
             3314:channel_request
              297:channel_request_i
               48:channel_assign
               23:channel_release
               22:session_close
                8:channel_release_i
                1:session_request
                1:connect_request

(b)    Graph::add_sequence( list<Graph>::iterator s ){

         if (s->symbol==”-stop”) return

         loop ( branches.begin to end){
            - find object with matching symbol as s->symbol
            - create and insert new object if needed
         }

         Graph* b= matching Graph object in branches list

         if b- >branches contains ”-skip” then{
             this- >add sequence( ++s )
             return
         }
         b->count++;
         b->add_sequence( ++s );
       }

(c)    main(){

         Graph message_graph

         list<Graph> code=[connect_request, connect_request_i, -skip, -stop]
         message graph.add sequence( code.begin() )

         loop over all devices{
            list<Graph> messages = sequential messages to/from device
            loop (m = messages.begin to end){
               message_graph.add_sequence( m )
            }
         }
         . . .
```

Figure 8.
Implementation of the "-skip" pruning node in the Graph class.

In Figure 9a, two -skip_r flags are inserted as shown. The re-aggregated graph is shown in Figure 9b. An additional -skip_r flag is added and the final graph is shown in Figure 9c. The simple rule for pruning at this point is to prune any internal messages.

Figure 9c shows the messages of interest that occur after the initial connect_request message. Line 9618 is the second message of the sunny day sequence and accounts for 98 percent of attempted connect_request messages. The remaining messages are the starts of various error legs.

```
(a)     9814:connect_request
              6307:channel_request_i
                                    -skip_r
              3314:channel_request
               100:channel_assign
                41:channel_release_i
                                    -skip_r
                24:session_close
                23:channel_release
                 2:connect_request
                 1:session_request
                 0:connect_request_i

(b)     9814:connect_request
              9577:channel_request
               100:channel_assign
                64:channel_release
                41:channel_reject_i
                                   -skip_r
                24:session_close
                 3:channel_response_i
                 2:connect_request
                 1:session_request

(c)     9814:connect_request
              9618:channel_request
               100:channel_assign
                64:channel_release
                24:session_close
                 3:channel_response_i
                 2:connect_request
                 1:session_request
```

Figure 9.
Pruning and aggregation of messages following the connect_request message.

The utility of the Graph algorithm is shown by comparing Figure 9a with Figure 9c. Figure 9c represents a condensed view of the information in the graph in Figure 9a, but which is not easy to extract due to the various other messages present.

Figure 9 also shows an iterative procedure for pruning the graph. Flags are added to the current graph printout and then the graph is re-aggregated and printed out below. Thus the flags that appear in a printout are used to create the next printout. This process proceeds iteratively until the graph printout contains the desired level of information.

Next the main sunny day connection scenario is followed to its conclusion using the same pruning technique, and then the first two error legs are investigated. **Figure 10** extends the main scenario of Figure 9c. The graph is pruned and extended to four levels deep as shown. Figure 10f represents the main sunny day connection scenario of Figure 1, which accounts for approximately 97 percent of initial connect_request messages. The failure rate at each message level can be calculated. Significant error legs are also seen, for instance, line 49 in Figure 10d.

Four new flag nodes are introduced to the graph in Figure 10f as shown in **Figure 11**. These flags will trigger histogram pegging in add_sequence() when the graph is re-aggregated. These flags are implemented in the add_sequence() routine similar to the –skip flag in Figure 8b. The flags will histogram the time difference between their head node and their head node’s head node. So for instance, -hist1 in Figure 11 will histogram the time between each message pair in lines 9814 and 9618. Timestamp and head pointer variables are added to the Graph class, and add_sequence() is modified to update the timestamp of each node when added/updated.

The resultant histograms are shown in **Figure 12**. For use in the design of the load generator, these histograms can be used as probability distribution functions (PDF) to determine the time spacing between messages received and sent from the load generator. These distributions are also useful for general system optimization and performance characterization. Various structures are exposed that come from internal system behavior. The tails measure the queuing delays inside the respective servers. Hist4 in Figure 12 is the connection duration distribution; the various sharp peaks come from autonomous behavior of the data devices or the switching system. The data collection period is 25 seconds and the true mean connection duration is approximately 10 seconds, so many of the channel_release messages are not captured.

Another common use of such timing distributions is in error investigation. Many problems have been first identified with such histograms. Software errors and design problems will often produce anomalies in such distributions. “Before” and “after” comparisons of these distributions around some event are also quite useful in practice.

```
(a)    9814:connect_request
             9618:channel_request
                   5645:channel_response
                   3881:channel_request
                                             -skip_r
                     91:channel_response_i
                                             -skip_r
                      1:channel_release

(b)    9814:connect_request
              9618:channel_request
                    9616:channel_response
                       1:channel_release


(c)   9814:connect_request
              9618:channel_request
                    9616:channel_response
                          5683:channel_assign
                          3925:channel_response
                                                 -skip_r
                             3:channel_release


(d)    9814:connect_request
              9618:channel_request
                    9616:channel_response
                          9560:channel_assign
                            49:channel_release


(e)    9814:connect_request
              9618:channel_request
                    9616:channel_response
                          9560:channel_assign
                               4281:channel_assign
                                                 -skip_r
                               3434:channel_release
                                 25:connect_request


(f)    9814:connect_request
              9618:channel_request
                    9616:channel_response
                          9560:channel_assign
                                6636:channel_release
                                  79:connect_request
                                   1:connect_request_i
```

Figure 10.
The main sunny day connection scenario.

Next the first two error legs of Figure 9c (lines 100 and 64) are explored. For purposes of brevity, the entire iterative graph/flag process will not be shown. The final result of this analysis on both error legs is shown in **Figure 13**.

The first most common error leg is shown starting at line 100. This error leg shows there is a toggling sequence of connect_request and channel_assign messages occurring for a small number of unique data devices. Further analysis done at Alcatel-Lucent

```
9814:connect_request
       9618:channel_request
                                                -hist1
             9616:channel_response
                                                -hist2
                   9560:channel_assign
                                                -hist3
                         6636:channel_release
                                                -hist4
```

Figure 11.
Instrumenting the graph of Figure 10f with "hist" flag nodes to collect local information.

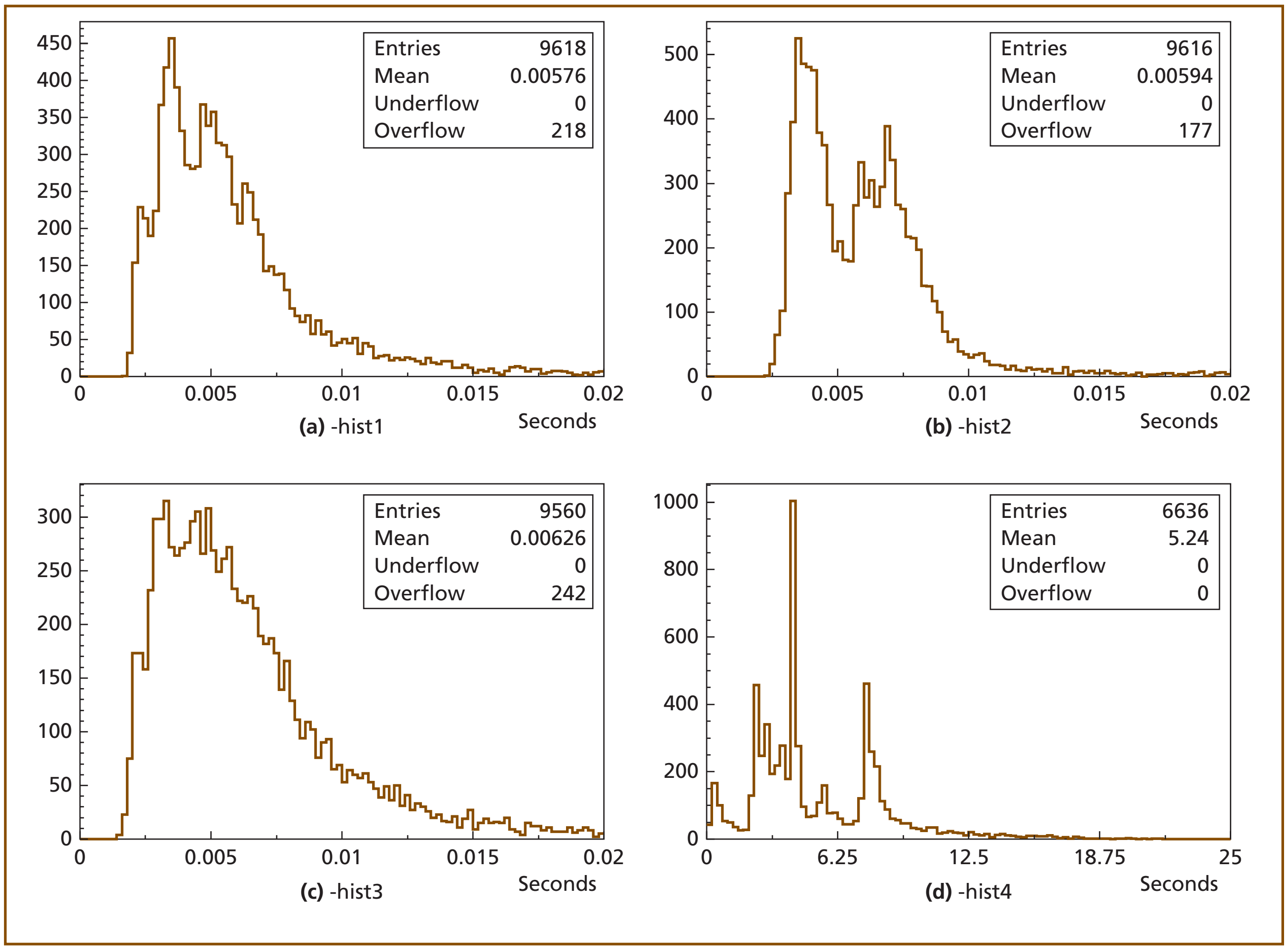

Figure 12.
Timing distributions as measured by the -hist flag nodes in Figure 11. These distributions measure the time difference between the two preceding messages in the graph relative to each flag.

```
9814:connect_request
      9618:channel_request
               ...

       100:channel_assign
               43:connect_request
                     41:channel_assign
                           24:connect_request
                           11:channel_release

        64:channel_release
               64:channel_request
                     64:channel_response
                           64:channel_assign
```

Figure 13.
The first two error legs of Figure 9c expanded and reduced out to a depth of 5.

suggests that this phenomenon occurs when the data device is in poor radio coverage. The channel_assign message is transmitted but not received by the data device. The data device then times out and resends a connect_request message.

Next, we explore the error leg in Figure 9c line 64. This appears to be a normal sunny-day connection sequence except that a channel_release message is inserted between the connect_request and channel_request messages. This sequence of messages can occur when a device has established a connection (i.e., received the channel_assign message from the signaling server) and then loses RF contact and starts a new connection by sending a connect_request message. In this case, the signaling server will tear down the existing connection before proceeding with the new connection.

For the purpose of this paper, we can stop with these two error scenarios. The final code graph is shown in **Figure 14**. The full analysis on the two error legs is shown. All of this code was declared iteratively as individual sequences of symbols.

The graph analysis provides detailed information about the most commonly occurring scenarios. The previously mentioned KPI counters provide the closest level of such information. However, the graph technique adds considerable detail to what can be discerned from KPIs only. KPIs are counts that are pegged at one or more places within the state machine code running on the signaling and data servers represented in Figure 1. This allows for general measurements of system performance. For instance, KPI counts exist to measure the percentage of all connect_request messages that successfully end in a connection setup (Figure 10f, lines 9814 and 9560). Many counts are also implemented to count error leg scenarios. While these counts are adequate to provide a high level view of the system, the graph analysis can show considerable further detail. For instance the toggling message sequence in Figure 13, line 100, had not previously been found using the KPI counts.

There is also considerable further detail available in the raw packets that each message is encoded in. In another graph analysis, a new smartphone had been introduced that had defects in the antenna design. By filtering on data device type, it was shown that the new device had discernible differences in its graph structure as compared to other devices due to this defect. This was shown within the first few days of device launch and helped to drive the problem resolution.

Other functions can be implemented as graph nodes. These functions are usually developed based on some analysis need. They are not covered here due to scope. Some other node types that have been developed include filter nodes that only permit certain symbols past a specified graph location, simple print flags to print some desired variable, and flags similar to the "hist" flags to aggregate some desired information in a node and then print it out with the node's destructor. Another useful technique is to build a graph with the sequences in reverse order; this allows the preceding messages before some event to be aggregated together and examined. Having an understanding of the relatively simple graph structure and its use allows complex datasets to be explored and useful information to be extracted with relative ease.

High Performance Load Generator

A load generator plays a critical role in the overall delivery of a successful switching system. There

```
0:message_graph
     21:connect_request
           1:connect_request_i
                1:-skip
           3:-skip_r
                1:channel_request_i
                1:channel_release_i
                1:channel_reject_i
           4:channel_release
                1:-skip_r
                      1:channel_release
                3:channel_request
                      1:-skip_r
                           1:channel_request
                      2:channel_response
                           2:-skip_r
                                 1:channel_response_i
                                 1:channel_response
           8:channel_request
                 2:-skip_r
                       1:channel_request
                       1:channel_response_i
                 5:channel_response
                       1:-skip_r
                             1:channel_response
                       3:channel_assign
                             1:-skip_r
                                   1:channel_assign
                             1:-hist3
                             1:connect_request
                                   1:-hist4
                       1:-hist2
                 1:-hist1
           5:channel_assign
                 1:channel_assign
                       1:-skip
                 3:connect_request
                      1:connect_request_i
                           1:-skip
                      2:channel_assign
                           1:channel_assign
                                 1:-skip
                           1:connect_request
                                 1:connect_request_i
                                       1:-skip
                 1:channel_release
                      1:channel_release
                           1:-skip
```

Figure 14.
The final code graph used in the analysis, including both error legs. This graph is built in main(), as in Figure 8c, prior to message aggregation and causes desired modifications of the graph to occur or local information to be extracted.

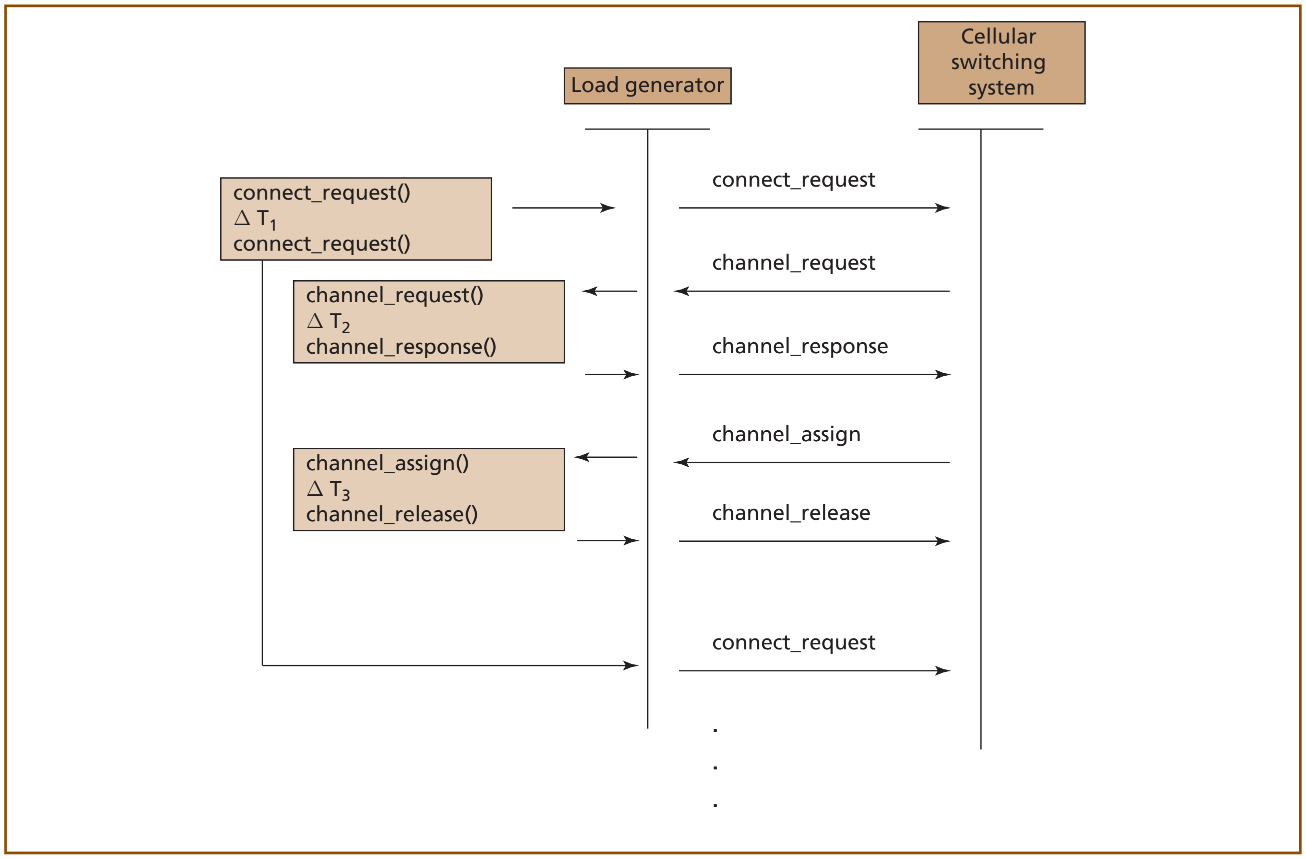

Figure 15.
The implementation of a sunny day connection scenario. Five functions (denoted with parentheses ()) are implemented within the data device class to send and receive the messages.

are many phases of this delivery, such as design, development, testing, deployment, and field support. In each of these phases the overall process is generally fixed, but there are many exceptions where specific details necessitate one-off test setups. Having a load generator that is well designed, offers high performance, is easy to use, and can be quickly configured for specific test cases is invaluable in all phases of system delivery.

Experience has shown that one of the most important factors in the design of a load generator (or for any software in general) is that the design be a simple as possible. The load generator will evolve over the life of the switching system, which can be tens of years, and an overly complicated initial design will not age well. Seemingly inconsequential design decisions early on can produce substantial negative impacts in the long term.

The load generator is implemented on a dedicated server which connects to the switching system as shown in **Figure 15**. A single process is used that contains cell tower and data device objects which will simulate the call model. Figure 15 shows the main sunny day connection sequence. It can be implemented by defining triplets of the form: [function1, ΔT, function2]. All call model scenarios will be implemented in this fashion.

A large number of data devices will be simulated, so a real time break must be taken between function1 and function2 to allow other objects to be serviced. This implies that a large number of timers will be running simultaneously in the load generator. To

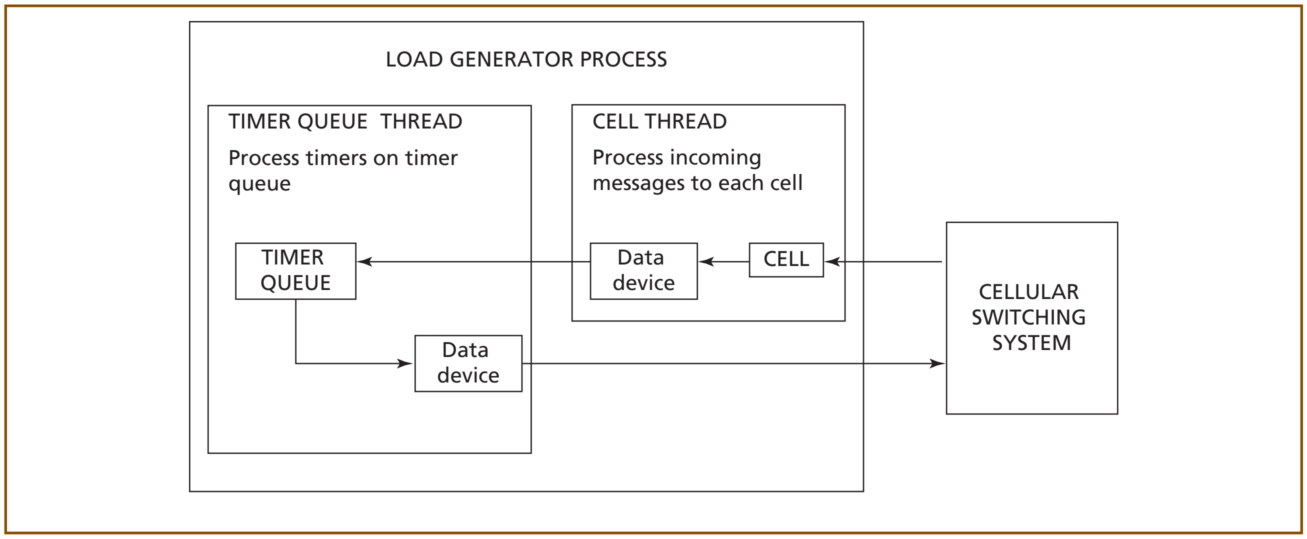
LOAD GENERATOR PROCESS
TIMER QUEUE THREAD
Process timers on timer queue
CELL THREAD
Process incoming messages to each cell
TIMER QUEUE
Data device
CELL
Data device
CELLULAR SWITCHING SYSTEM

Figure 16.
Schematic diagram of the load generator showing the processing of a [function1, ∆T, function2] triplet.

implement this, a timer object is defined. This object has a list of running timers. A member function will spin on this list of timers and call a specified callback function on expiry of each timer. A thread is spawned to execute this code. This thread is locked to a core and allowed to run uncontested.

A second thread type is defined that will service the incoming messages in Figure 15. These messages are exchanged through ports opened by the cell objects. This thread loops over these cells and services the incoming packets on all of the ports. Received messages are decoded and the appropriate object/function pair is invoked to handle the message.

Figure 16 shows a schematic representation of this basic design. An incoming message is decoded by a cell object called from the cell thread. A data device function for the specific message is called to process the message. This function determines that the next step will be to wait some period of time and then send a return message. It inserts a timer with a callback function into the timer object that services the data device and gives up program control. When the timer expires, the timer object calls the callback function to format and send the reply message. The reply message is sent directly from the data device object in the timer queue thread. The data device object is assigned a cell object, which it uses to route the reply message.

The declared cell and data objects are assigned to these two thread types. More or fewer objects can be assigned to these threads as performance allows. The two thread types are then replicated across available cores to fully utilize the server. Each thread is locked to a core and allowed to run uncontested. A separate thread is also started which uses the CERN ROOT package [3] for program input/output (I/O).

Poisson timing between the connections (ΔT_1) is generated from a Gaussian PDF as in **Figure 17**. Likewise, the measured timing distributions from Figure 12 can also be used as the input PDF for generating the other timings (ΔT_2, ΔT_3).

The process of implementing specific call model scenarios is then one of defining sets of the [function1, ∆T, function2] triplets. As with the preceding discussion of code based analyses, there is typically more than one way to implement a scenario; here trying to find the simplest implementation is generally preferable. Some implementations can be challenging, generating sometimes unexpected results. The general procedure is to monitor the generated messaging sequences for the expected behavior. The

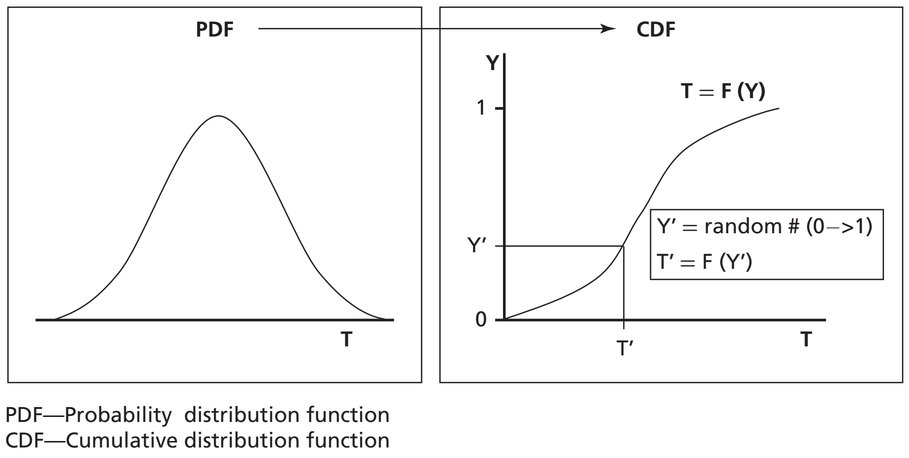
PDF
CDF
T
Y
1
T = F (Y)
Y′
Y′ = random # (0–>1)
T′ = F (Y′)
0
T′
T
PDF—Probability distribution function
CDF—Cumulative distribution function

Figure 17.
Generating a distribution of time values T from a known PDF. The CDF can be analytically derived from the PDF if possible, or numerically integrated from a measured distribution.

message graph is very useful in this debugging phase. A further simplification is to build a graph within the load generator process based on internal sequence data. This yields similar insights while eliminating the intermediate steps of data capture and pre-processing. Once a call model is stable, the histogram flags can be used in the internal load generator graph as a high precision timing measure of switching system performance (as in Figure 12). Software defects have often been first detected as anomalies in such distributions.

A prototype version of the load generator design described above is implemented on two cores of a single server running at 1.3 GHz. The generator is capable of simulating one million connection attempts per hour. The cell and timer queue threads each run at 100 percent occupancy on their assigned cores, spinning on their respective tasks. The time spent processing messages outside of the spin loop is measured at approximately five percent. As this out-of-loop percentage is increased it will induce messaging latency as messages get queued waiting to be processed. This is generally undesirable and can produce second-order effects like message retries from the switching system. The exact amount of tolerable latency has to be determined empirically.

This load generator design has been implemented for the Alcatel-Lucent third generation (3G) EV-DO system. This system is nearing end-of-life as data devices migrate to the fourth generation (4G) Long Term Evolution (LTE) standard. The amount of future work on this specific load generator will thus be limited, however we hope that the lessons learned can be transferred to future systems.

Optimizing Pre Release Software Testing

For the purpose of designing and optimizing the pre-release software testing program for a cellular switching system, it is helpful to examine some basic statistics on software failures. For the Alcatel-Lucent EV-DO system, the signaling server process represented in Figure 1 has approximately 1.5 million lines of C/C++ code and the post release core file failure rate is approximately one per week per 15000 instances. On average, each instance processes approximately one million connections per day. Each failure is examined and eventually fixed. Pre-release software testing of the same process consists of approximately 300 instances run under varying test lab loads throughout a six month development cycle. The deployed load on this process is thus considerably larger than is possible in the test labs.

Many of the software failures are related to non-thread-safe code. Failures such as these occur when a specific section of code is interrupted under some precise set of circumstances which usually occur very rarely. We can conclude that even if the load generator could exactly replicate the live system connection load, it would not be sufficient to find all, or even most, software faults given the much larger number of deployed instances. Thus, a decision is needed on how to make the best use of available testing resources during the development cycle. This resource allocation problem is solved iteratively. Having a flexible load generator that can be adapted to ongoing empirical experience is of the utmost importance. Human factors are also critically important. Being able to quickly configure and execute tests based on some important issue at hand can be critically important to the success of a testing program.

Conclusion

A second generation design of a load generator for testing cellular switching systems is presented. The design is focused on simulating scenarios seen in a live system generated by real data devices. The scenarios are measured using an IP packet capture of the messaging between the cell towers and the switching system. This data is analyzed with an iterative graphing technique that allows for the most common scenarios to be isolated and studied. Specific scenarios are added to the load generator by implementing functional triplets of the form [function1, wait time, function2].

Acknowledgements

Many people have been involved with the work described in this paper. With high confidence, it can be assumed that almost everyone involved with the Alcatel-Lucent EV-DO system has contributed in some way. The leading figures are Steve Sommars, Randy Pitt, Kusum Singh, Camille Sobers, Gary Hurm, Steve Samp, Tom Chlapecka, Paul Rietz, and the late Sal Suero.

The author would also like to thank the reviewers and editors for help in preparing the manuscript.

*Trademark

Wireshark is a registered trademark of the Wireshark Foundation, Inc.

References

[1] Alcatel-Lucent, "Alcatel-Lucent 9900 Wireless Network Guardian," White Paper, Dec. 2012, <http://www.alcatel-lucent.com/products/9900-wireless-network-guardian>.

[2] F. P. Brooks, Jr., The Mythical Man-Month: Essays on Software Engineering, 2nd ed., Addison-Wesley Longman, Boston, MA, 1995, Chapter 7.

[3] R. Brun and F. Rademakers, "ROOT—An Object Oriented Data Analysis Framework," Nuclear Instruments Methods Phys. Res. A, 389:1-2 (1997), 81–86.

[4] A. Buvaneswari, L. Drabeck, N. Nithi, M. Haner, P. Polakos, and C. Sawkar, "Self-Optimization of LTE Networks Utilizing Celnet Xplorer," Bell Labs Tech. J., 15:3 (2010), 99–117.

[5] D. Loukatos, L. Sarakis, K. Kontovasilis, and N. Mitrou, "An Efficient ATM Traffic Generator for the Real-Time Production of a Large Class of Complex Traffic Profiles," J. Commun. Networks, 7:1 (2005), 54–64.

[6] U. Niranjan, G. P. Ramacharyulu, and K. Vijaya, "Packet Load Generator for Telecom Networks," Internat. J. Adv. Res. Comput. Eng. Technol., 2:5 (2013), 1720–1724.

[7] Wireshark Foundation, <http://www.wireshark.org/>.

(Manuscript approved November 2013)

KENNETH W. DEL SIGNORE is a member of the technical staff at the Alcatel-Lucent corporate campus in Naperville, Illinois. He is one of a class of approximately fifty former high energy particle physicists that came to Alcatel-Lucent from Fermi National Accelerator Laboratory in the mid-to-late 1990s. His previous work has included several patents for paging algorithms that are used to reduce RF signaling overhead in the Code Division Multiple Access (CDMA) system. His current work includes performance characterization and optimization on Alcatel-Lucent's EV-DO and 1×CDMA Wireless Management System (WMS) switching systems. ◆