View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KU ScholarWorks

Instrumentation and Evaluation of Distributed Computations

By

William Dinkel

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

____

Chairperson Dr. Victor Frost

____

Dr. Arvin Agah

____

Dr. Prasad Kulkarni

Date Defended: July 18th, 2013

The Thesis Committee for William Dinkel

certifies that this is the approved version of the following thesis:

Instrumentation and Evaluation of Distributed Computations

____

Chairperson Dr. Victor Frost

Date approved: July 18th, 2013

Abstract

Distributed computations are a very important aspect of modern computing, especially given the rise of distributed systems used for applications such as web search, massively multiplayer online games, financial trading, and cloud computing. When running these computations across several physical machines it becomes much more difficult to determine exactly what is occurring on each system at a specific point in time. This is due to each server having an independent clock, thus making event timestamps inherently inaccurate across machine boundaries. Another difficulty with evaluating distributed experiments is the coordination required to launch daemons, executables, and logging across all machines, followed by the necessary gathering of all related output data. The goal of this research is to overcome these obstacles and construct a single, global timeline of events from all servers.

We employ high-resolution clock synchronization to bring all servers within microseconds as measured by a modified version of the Network Time Protocol implementation. Kernel and user-level events with wall-clock timestamps are then logged during basic network socket experiments. These data are then collected from each server and merged into a single dataset, sorted by timestamp, and plotted on a timeline. The entire experiment, from setup to teardown to data collection, is coordinated from a single server. The timeline visualizations provide a narrative of not only how packets flow between servers, but also how kernel interrupt handlers and other events shape an experiment's execution.

Acknowledgments

None of this research would have been possible without the guidance and wisdom of Dr. Douglas Niehaus. This work is dedicated to his memory. Rest in peace, Doug.

A heartfelt thanks goes out to all of his students that contributed over the years to each piece of the KUSP puzzle.

To Dr. Frost and the faculty and staff at the Department of Computer Science and Electrical Engineering, I am deeply appreciative of their willingness to stick with me all these years and allow me to finally complete this work. Pamala Shadoin, you deserve an award!

And to my wife, Jess, who lived with the ups and downs, fits and starts, of this whole process. You're my rock.

Table of Contents

Abstract....iii
Acknowledgments....iv
Table of Contents....v
List of Figures....vi
List of Programs....vii
1 Introduction....1
2 Related Work....5
  2.1 Clock Synchronization....5
  2.2 Instrumentation....6
  2.3 Automation....7
  2.4 Visualization....7
3 Implementation....9
  3.1 Clock Synchronization....9
  3.2 Instrumentation....13
  3.3 Automation....17
  3.4 Post-Processing and Visualization....23
4 Evaluation....28
  4.1 Clock Synchronization....28
    4.1.1 Clock Synchronization Daemon....28
    4.1.2 Regression Test....30
  4.2 Post-Processing and Visualization....33
5 Conclusions and Future Work....39
6 References....40

List of Figures

1 NTP Packet Flow....10
2 Interrupt buffering causing NTP timestamp delay....12
3 DSUI event pairing....17
4 NetSpec Experimental Flow - Generalized....19
5 Clock synchronization regression test - NAPI on (client to server)....31
6 Clock synchronization regression test - NAPI on (server to client)....31
7 Clock synchronization regression test - NAPI off (client to server)....32
8 Clock synchronization regression test - NAPI off (server to client)....32
9 NetSpec Experiment Flow - 3 server socket pipeline....34
10 Global timeline plot - Full....36
11 Global timeline plot - DSUI event start....37
12 Global timeline plot - out-of-order event sequence....38

List of Programs

1 Modified NTP data structure....11
2 net_rx_action() with DSKI events....16
3 NetSpec DSKI sub-daemon control file entry....22
4 NetSpec distributed computation sub-daemon control file entry....22
5 NetSpec scoreboard schedule example....23
6 DSUI post-processing control file: per-server sort....25
7 DSUI post-processing control file: global timeline filter....26
8 Clock synchronization daemon output....29
9 Event pairs - 3 server socket pipeline....34
10 Global timeline post-processing console output....35

1 Introduction

Distributed computing is an area of significant research focus, given its ubiquitous use in such popular applications as web search, massively multiplayer online games, financial trading, and cloud computing [1]. Each of these applications consist of computations that can be broken down and distributed to many individual servers, where each server completes a portion of the work. It is a proven model, as evidenced by the everyday, global importance of the above applications.

Thorough evaluation is required in order to ensure that these distributed computations are performing efficiently and correctly. Monitoring packages are available that provide simple statistical measures such as run time, average server CPU load, memory and disk usage, and network utilization [2]. This information can be used to isolate underperforming systems and locate bottlenecks, but it does not provide sufficient detail to diagnose what operating system events may be causing processing delays, especially if these delays propagate throughout the system. What is needed is a method to generate an individual accounting of events that occur on each server during the computation. Constructing a single, global timeline of all events from each server would allow for a complete evaluation.

There are many difficulties with generating such a global timeline. Each event must be timestamped to allow for proper sorting from start to end. Most distributed systems are built with servers that contain commodity-off-the-shelf (COTS) hardware, and run a standard operating system such as Linux [3]. No two server processors are exactly the same and each will have its own unique tick rate. Thus each computer's clock will eventually deviate [1]. This deviation makes it impossible to properly sort events from different servers into one timeline. While the

Network Time Protocol (NTP) service within the Linux operating system can synchronize each server to a global clock, it is limited to within millisecond resolution and requires that frequency adjustments be made in small increments over time, taking several hours to become synchronized [4]. Finer resolution is required to accurately compare sequences of system events, especially those within the kernel. One-time frequency adjustments are preferable, especially in an experimental setting where system reboots or power-down sequences are common, requiring re-synchronization. Reducing this synchronization process to a matter of minutes rather than hours allows for more time to run distributed experiments. Knowing these limitations, our approach is to modify NTP and the Linux kernel to improve clock synchronization resolution to within microseconds and allow immediate frequency adjustments.

Logging execution trace data for the global timeline events is challenging, in that their frequency of occurrence, especially events within the kernel, can lead to excessive storage requirements and processing delays. It is highly desirable for the event-logging process to be lightweight and unobtrusive, so as not to take server cycles from the distributed computation and inadvertently skew the results. Echoing this trace information to the system console with *printk* and *printf* statements is a commonly used method, but is costly and can sometimes lead to message loss [5]. What is required is a facility to store data into memory as events occur, with the ability to process the data to persistent storage either under operating system scheduler control during the experiment or offline later. We use the Data Streams Kernel Interface (DSKI) and User Interface (DSUI) to instrument both the kernel and user code. These interfaces allow run-time enabling of desired trace events and log the data to memory buffers during the experiment. They also provide daemons that run during the experiment to process the data from memory to a system file.

Another difficulty is coordinating the setup, execution, and cleanup of the distributed computation across all servers. First, the server clocks must be synchronized. Then all trace events must be enabled and collection daemons started. Third, the distributed executable must be launched. Finally, upon completion of the distributed experiment, all trace output files must be gathered and collection daemons shutdown. Performing each of these tasks manually via a network console such as *ssh*, even on as few as two servers, would be a timely and error-prone process. This is alleviated by the use of NetSpec, a utility that supports the description and execution control of arbitrary distributed computations. Each of the experiment phases described above can be specified in a NetSpec control file. Specialized NetSpec daemons are run on each server, which listen for messages coming from the NetSpec controller process running on a central machine. The controller coordinates the beginning and end of each phase across all machines, automatically executing the distributed computation.

The final challenge in constructing a global timeline is presenting the trace data in a visual format that can accurately convey the flow of information between servers. Once all data has been collected, it is a simple process to sort it into one dataset by event timestamp. We then use the *GNUplot* utility's X/Y plot in a novel fashion. Each event is plotted with time on the X axis increasing from left-to-right. Each server in the computation is assigned two adjacent Y-values: one for its kernel events and another for its user events. Thus all events that occur on a server will be plotted on the same row. Additional tag information logged with the events allows for relational pairs to be formed between events on different servers, such as network send and receive calls. These event pairs have lines drawn between them. Thus the graph can be read left-to-right, following the lines up and down between rows, illustrating information flow between the servers during the execution of the distributed computation.

The rest of this paper is organized as follows. Chapter 2 details related work in the fields of clock synchronization, instrumentation, automation, and global timeline visualization. Chapter 3 presents the implementation details for the proposed solution in each of the aforementioned fields. Chapter 4 presents the experimental testing process and results. Finally, chapter 5 discusses any conclusions that can be drawn from the solution and possibilities for future work.

2 Related Work

While timeline visualization of specialized distributed computations has been previously reported [16-17], visualization of a global timeline for arbitrary distributed computations is a unique concept. Other solutions to the component problems of clock synchronization [6-10], instrumentation [11-13], and automation [14-15] have been proposed as well. A complete discussion must mention these works, and is presented in the following sections.

2.1 Clock Synchronization

In addition to the Network Time Protocol, several other algorithms and protocols have been developed to address clock synchronization. Clock Sampling Mutual Network Synchronization (CS-MNS) is a non-hierarchical and mutual network synchronization algorithm for wireless and ad hoc networks [6]. It is shown to synchronize server nodes to within 100s of microseconds via the use of specialized wireless hardware and the TinyOS operating system [7]. This resolution is within the range required to instrument distributed computations, but the use of wireless is not ideal for servers in a data center.

The Precision Time Protocol (PTP) is based upon the IEEE 1588 standard that provides sub-microsecond clock resolution over a local area network [8]. It is specifically designed to provide accuracy beyond that achieved by NTP. It calculates time offsets by measuring send and receive timestamps between a master and all slave servers. These offsets are compensated for by speeding up or slowing down the system clock. This implementation is very much similar to our proposed solution, and as such is subject to the same limitations of processing delays that can be caused by the kernel [9]. Our approach addressed these delays by modifying the kernel network

stack directly. At the time our global timeline research was performed, implementations of the PTP standard on Linux were in the experimental stages, and did not modify the kernel network stack. This implementation has since matured and been merged into the mainline Linux kernel, addressing these sources of delay, and is a viable alternative to the modified version of NTP used in this research [10].

2.2 Instrumentation

The instrumentation of code, both kernel and user-level, is a diverse field with several solutions, especially for the Linux operating system. SystemTap is an industry-supported tool that provides a preset number of built-in events within the kernel [11]. These events can be enabled via scripts, with custom processing hooks that execute with each event occurrence. Data can be collected at both the kernel and user level, although there are limitations with extracting some data from user-space [12]. With the use of static probing markers within the kernel, instrumentation overhead is kept to a minimum.

The Linux Trace Toolkit next generation (LTTng) is a kernel and user-space tracer designed to have a low impact on performance [13]. It provides both markers for quick instrumentation during debugging and tracepoints for more permanent instrumentation. Events are stored in memory in ring buffers, and collected by a daemon during execution. Event processing overhead is at the sub-microsecond level.

Each of the aforementioned solutions provides the instrumentation facilities to evaluate distributed computations. We selected the DSKI and DSUI as our platform because of their existing integration with the NetSpec utility. With a NetSpec control file, we could specify which logging events to enable at run-time, control the start and stop of the collector daemons, as well

as manage the movement of all output files.

2.3 Automation

Automating the execution of distributed computations is specialized, mostly relevant to the task at hand. In our research, NetSpec, an end-to-end network testing tool, was found to be ideally suited for the task of constructing global timelines, as it was designed with instrumentation and data collection as part of the requirements [14]. The strongest parallel in the field can be found in the Message Passing Interface (MPI), a communications protocol for parallel computers whose implementations include utilities that allow for launching a distributed computation across many independent server nodes [15]. In most implementations, the *mpirun* command is essentially a wrapper around the *rsh* or *ssh* remote shell utilities, which will stage the computation executable to each server from a master. Each server enters a wait-state until a signal is sent from the master to proceed. This is very similar to NetSpec, in that it coordinates the experiment in phases, and servers wait for the next "phase start" message from the NetSpec controller before proceeding. The drawback with MPI and its implementations is that it is not general-purpose and provides no interfaces for coordinating other daemons outside of the distributed computation. Thus clock synchronization and code instrumentation would need to be handled manually by other scripts outside of MPI.

2.4 Visualization

Several packages are available to visualize the results of a distributed computation trace. LTTng includes the Linux Trace Toolkit Viewer (LTTV), which is a graphical interface for visualization of LTTng trace data [16]. The output from each process in the trace is rendered on a row in the graph, much like each server is rendered on a row in our GNUplot solution. The

LTTV also presents other detailed trace information such as process information and statistics for event types. It also allows for interactive zooming based on start and endpoint times.

For visualization of MPI application executions, the Jumpshot tool can generate a graphical representation of all MPI-related calls, plotting their occurrence on a timeline [17]. As with LTTV, each server trace is plotted on its own row in the graph. Lines can also be drawn between events on different rows, representing related MPI send and receive calls between servers. This view of the progression of the computation and how it “travels” across servers is much like that envisioned for our global timeline. The only limitation is that Jumpshot is restricted to MPI calls, and not any arbitrary instrumented event.

As these solutions are dependent upon their parent packages, the research presented in this paper elected to use a simpler, customized approach with the commonly-available GNUplot tool. As a widely-used graphing tool in research and academia, it provided the necessary features to produce a compelling global timeline rendering.

3 Implementation

Our proposed solution for constructing global timelines is built upon work in four separate fields, each detailed in separate sections below. The first three sections, covering clock synchronization, instrumentation, and automation, are all components involved in the execution of the distributed computation. The majority of this work is an extension of research performed previously by students at the University of Kansas Information and Telecommunications Technology Center. The last section details new work developed to post-process and visualize the resulting instrumentation data to produce a global timeline.

3.1 Clock Synchronization

The NTP distribution is widely-used to synchronize servers to a central time server and can achieve accuracies of less than a millisecond on Local Area Networks (LANs) [18]. Its algorithm for computing time offsets and clock adjustments is well documented and shown to be accurate on stable networks [19]. On the Linux operating system, what prevents NTP from finer accuracy is in how the implementation runs almost entirely in user-space. The only kernel-level calls made by the NTP daemon are the system calls provided to get and set system time. The daemon performs calculations based upon data sent in network messages. The kernel must process these messages through the network stack, and at any time context can be switched away to handle another system interrupt. There is also no guarantee that the kernel scheduler will switch context to the NTP daemon immediately upon leaving kernel-mode. These are sources of delay that result in reduced accuracy when calculating time offsets. NTP can compensate for these delays, but only to a limited extent.

Rather than reinvent the calculations that NTP performs, our solution is to attempt to eliminate these sources of delay and provide more accurate timestamps. Figure 1 illustrates the flow of synchronization packets between a client and server. Timestamps are taken on the client at times T0 and T3, and on the server at times T1 and T2. These values are used to calculate an offset. The accuracy of this offset is dependent upon the amount of processing time that elapses between physical packet arrival and the actual handling of the message by the NTP executables running in user-mode. These executables will not make the system calls to retrieve the current time until context has switched to them. This could be delayed by the kernel handling subsequent interrupts and/or the operating system scheduler selecting a different user process to run. The solution is to retrieve the current timestamp within the kernel *as soon as possible after* the packet arrives.

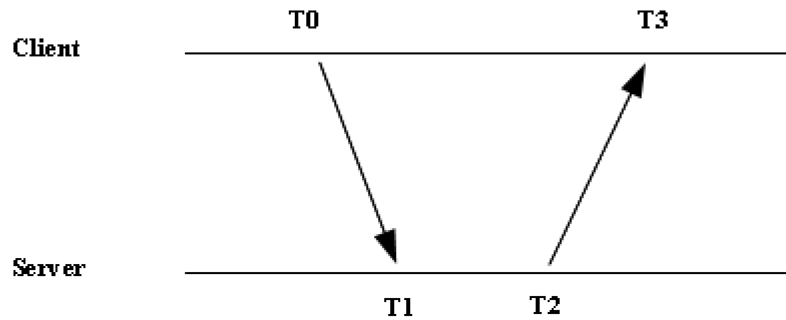

Figure 1: NTP Packet Flow

We build upon previous clock synchronization modifications made to the NTP subsystem and the Linux kernel network device drivers and network interrupt handler [20]. These introduced additional timestamp fields into the standard NTP packet structure, named *start_ts, rx_ts, tx_ts,* and *end_ts,* as shown in Program 1. An additional field, *magic_num*, is added to the structure as well. When synchronization packets are created by the NTP client and server

daemons, this *magic_num* field is populated with a unique ID. When a packet is received during packet processing by the kernel network interrupt handler, a check can be made against this field. If the ID matches that of the known “magic” number value, we are guaranteed that the current packet is a an NTP synchronization packet. An immediate timestamp is retrieved and inserted into the packet's *rx_ts* field, denoting the time at which the NTP packet was received. For the sending case, the same magic number can be checked for in the network device driver, just before the packet is queued in the send buffer. If found, the current timestamp will be inserted into the packet's *tx_ts* field. With these timestamps already recorded in the packet, the NTP user-mode utilities no longer need to request delayed system timestamps in user-mode. They can proceed with the offset calculation using more accurate data.

```
struct pkt {
        u_char  li_vn_mode;     /* leap indicator, version and mode */
        u_char  stratum;        /* peer stratum */
        u_char  ppoll;          /* peer poll interval */
        s_char  precision;      /* peer clock precision */
        u_fp    rootdelay;      /* distance to primary clock */
        u_fp    rootdispersion; /* clock dispersion */
        u_int32 refid;          /* reference clock ID */
        l_fp    reftime;        /* time peer clock was last updated */
        l_fp    org;            /* originate time stamp */
        l_fp    rec;            /* receive time stamp */
        l_fp    xmt;            /* transmit time stamp */

#ifdef NEW_NTP_CHANGES
        u_int32 magic_num;
        unsigned long long start_ts;        /* start field added at KU */
        unsigned long long rx_ts;           /* rx field added at KU */
        unsigned long long tx_ts;           /* tx field added at KU */
        unsigned long long end_ts;          /* end field added at KU */

        long               xtime_tv_sec;
        long               xtime_tv_nsec;
        unsigned long long xtime_tsc;
        unsigned long tsc_khz;

        unsigned long saddr;
        unsigned long daddr;
        u_int32 pkt_id;         /* unique packet identifier added at KU */
```

Program 1: Modified NTP data structure

The previous implementation's Linux kernel was a version that routed all network

devices through the same interrupt handler when messages were received. This handler performed common network stack processing before handing control over to any device-specific routines. Unless interrupts were masked, it was a reliable assumption that the handler would be called soon after arrival of any packets, and that the timestamp would be retrieved and inserted into the NTP synchronization packet shortly afterwards. Since that original research was performed, computer architecture has improved and the Linux kernel has expanded to allow per-device network interrupt handlers. With each device driver implementing its own handler, advanced features such as interrupt buffering/coalescing can be introduced. This feature allows the network device to postpone processing the receive queue until a preset number of interrupts have occurred. This reduces the amount of time the kernel spends locked in interrupt context while processing packets from the queue.

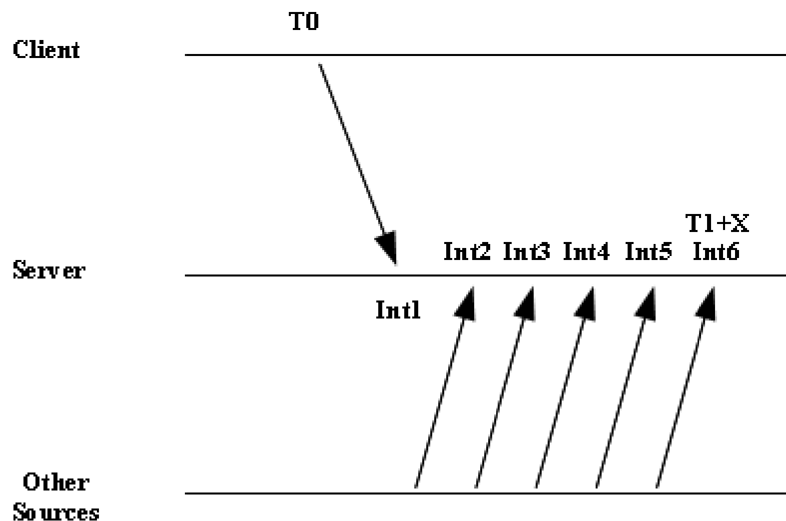

Figure 2: Interrupt buffering causing NTP timestamp delay

This interrupt buffering can be beneficial for general purpose systems, especially those that receive many small, separate messages. Instead of each transmission having to be processed under interrupt context, the kernel will only need to lock once to process a larger batch. Unfortunately, this causes complications for NTP timestamp insertion. With interrupts being

buffered, an NTP packet may arrive with one of the earlier interrupts. Its processing could be delayed until the predetermined number of interrupts has been met. This phenomenon is illustrated in Figure 2, where the timestamp is not taken until time $T1+X$, with X being the time between the first and sixth interrupts, assuming the device driver was waiting for 6 interrupts to occur before processing the queue. This problem results in less accurate time offsets being measured by NTP, and increases the likelihood that the timer adjustments made will skew the clock further than intended.

The test systems for this research utilized Intel e1000 Gigabit Ethernet network interfaces. This device's Linux driver implemented interrupt buffering in a feature called “RX Polling (NAPI)”. As this feature could not be disabled, a workaround was implemented in the driver's interrupt handler code to bypass the NAPI-specific handling during periods of clock synchronization. Specifically, the clock synchronization daemon sets a kernel state variable signifying that the system is actively synchronizing. Later, when the e1000 interrupt handler is called in response to a packet arrival, the receive queue is immediately processed, allowing for immediate insertion of the NTP timestamp. When the clock synchronization daemon exits, usually at the end of the distributed experiment, the kernel state variable is unset. Thus the NAPI functionality is restored, allowing the driver to function as intended for general system use.

Chapter 4 provides detailed experimental results for clock synchronization both with interrupts buffered and with the buffering disabled.

3.2 Instrumentation

The Data Stream Kernel and User Interface (DSKI and DSUI) are subsystems that support gathering system behavioral data from instrumentation points [21]. These points are

macros inserted into the kernel and user-level source code. When a thread of execution crosses a point, a data structure is generated and stored into kernel memory, to be moved to persistent storage later by a separate logging process. The types of points supported are counters, histograms, and events, with each type saving a specific set of data. Counters simply increment a variable with each occurrence. Histograms take a value and update a set of metrics that describe a histogram (*min, max,* and the number of occurrences for each bucket). The event type, though, is of particular use for distributed computations. This type allows logging of extra user-specified data, whether a number or an entire data structure, along with the current system timestamp. After a distributed computation has executed, the accumulated set of events can sorted by these timestamps, and then used to construct a global timeline. The extra data included with the events can be used to provide additional detail in the timeline as well.

Choosing the right set of events to log during the experiment is especially important. The logical place to start are those sections of the code dealing with network transmission, as this is where the flow of the computation between servers can be observed. In the user executables, DSUI events can be logged just before a packet is sent and immediately after a packet is received. The kernel provides many more potential instrumentation points within the network stack. Several example locations include:

- Device driver interrupt handler
- Device driver send/receive methods
- Soft interrupt send/receive methods
- IP layer processing methods
- TCP layer processing methods

Enabling all of these DSKI events can produce a very detailed view of how a system steps through the network subsystem. In the early stages of global timeline construction, all of these events were enabled. What soon became evident was that their volume and repetitiveness was difficult to visualize effectively on a timeline without being too cluttered.

Other per-system kernel information, such as system scheduler context switch events and other system interrupt handler processing, could provide additional detail into the computation. What was found through experimentation, though, was that the volume of these events caused the dataset to become far too dense. This was especially evident when the scheduler context switch events were enabled, as they were logged for *every* process switch, not just for those processes involved in the distributed computation. The Linux scheduler can potentially run every 4ms, which could generate hundreds of context switch events per second. Filtering through these events to locate network-related events, especially in a visual representation, was time-consuming and counter-intuitive.

In the end, the most benefit was found by instrumenting all user-level send and receive calls, as well as the start and end of the kernel's *net_rx_action()* routine. Illustrated in Program 2, this routine is the common network receive soft interrupt. It signifies the point where the kernel processes network messages, sending them up the stack to user-space. The routine steps through each available network device that is ready to process its receive queue. Note how NAPI is mentioned. As discussed in section 3.1, our modifications to the e1000 device driver will force this processing to occur every time on this device, so as not to delay NTP synchronization packet timestamping.

```
static void net_rx_action(struct softirq_action *h)
{
        ...

        while (!list_empty(list)) {
                struct napi_struct *n;
                int work, weight;

                ...

                n = list_entry(list->next, struct napi_struct, poll_list);

                have = netpoll_poll_lock(n);

                weight = n->weight;

#ifdef CONFIG_KUSP_DSKI_NET
                DSTRM_EVENT(NET_DEVICE_LAYER, NET_RX_ACTION_START, 0);
#endif

                /* This NAPI_STATE_SCHED test is for avoiding a race
                 * with netpoll's poll_napi().  Only the entity which
                 * obtains the lock and sees NAPI_STATE_SCHED set will
                 * actually make the ->poll() call.  Therefore we avoid
                 * accidently calling ->poll() when NAPI is not scheduled.
                 */
                work = 0;
                if (test_bit(NAPI_STATE_SCHED, &n->state)) {
                        work = n->poll(n, weight);
                        trace_napi_poll(n);
                }

#ifdef CONFIG_KUSP_DSKI_NET
                DSTRM_EVENT(NET_DEVICE_LAYER, NET_RX_ACTION_END, 0);
#endif

                ...
        }
}
```

Program 2: net_rx_action() with DSKI events

In order to logically connect events between servers in the global timeline visualization, such as a send/receive sequence, additional information must be added to establish a pair-wise relationship. This was accomplished by formatting a unique identifier string consisting of the user-level process' ID and the current count of messages sent. These strings would be stored in the extra data field of the DSUI send and receive events of the respective servers' user-level processes. For example, the 5th message sent from process '1' on server A would save the string "000010000000005" into the DSUI send event's extra data field. Server A would then include this string in the packet sent to Server B. Upon receipt, server B would strip this string from the

packet and save it in the extra data field of the DSUI receive event. This is illustrated in Figure 3. Using the message count as part of the identifier guarantees that each pair of events will have a unique identifier, and can be connected via a line in the visualization.

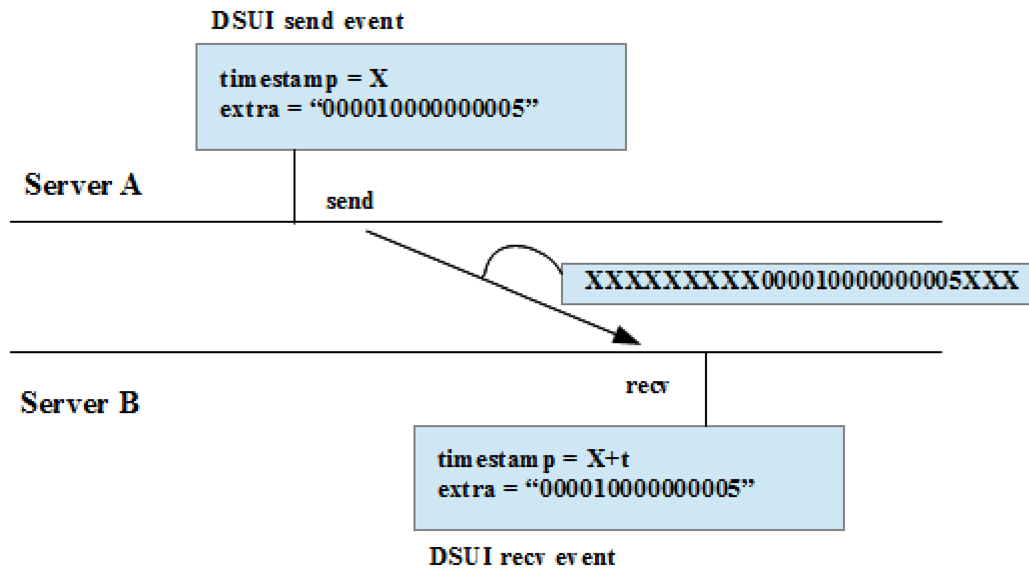

Figure 3: DSUI event pairing

3.3 Automation

NetSpec is a network performance evaluation tool. It was originally designed to coordinate the execution of end-to-end network tests, with the ability to create hundreds of network flows across tens of nodes. The nodes could be passive, taking only measurements, or active, and generating traffic. It has been used in research projects to simulate and evaluate traffic patterns on a variety of network hardware. Distributed computations fit into this same category, as the network is an essential piece to the puzzle. Thus NetSpec was an ideal utility for the automation of global timeline experiments.

A NetSpec distributed computation involves a series of executables, hereafter referred to

as *sub-daemons,* running on an arbitrary set of network-connected machines, hereafter referred to as *slaves*. These sub-daemons can perform any form of work, whether it be confined to the local machine or over the network to the other slaves involved in the current computation. All sub-daemons are launched at the beginning of the experiment by the primary NetSpec daemon, *netspecd*, an instance of which is running on each slave. The command to launch *netspecd* itself, as well as the subsequent sub-daemon launches, comes over the network from the *ns_control* program. This is the NetSpec master process, launched by the user from a machine on the same network as the slaves. It can run from an independent machine or one of the slaves itself.

Rather than launch the sub-daemons during the experiment just before they are meant to perform work, NetSpec enforces the policy that all sub-daemons must exist as a process on the slave and in a *wait* state before the experiment officially begins. This allows for *phase*-based execution, in that individual sub-daemons can divide their work into different phases. They awaken at phase-start and re-enter a wait state at phase-end, but do not terminate until the end of the experiment. This allows state information to be preserved across phases. The *ns_control* process sends the phase "start" messages to the sub-daemonss.

The timeline for this phase-based execution is specified by a *scoreboard* schedule consisting of tuples of sub-daemons and their associated phases. These tuples are grouped into blocks. At the start of the experiment (once all of the sub-daemons are confirmed to be in a wait state), each phase specified in the first block is started. When *ns_control* receives successful phase completion messages from every phase in the first block, it proceeds to the next block, starting the phases specified within. This process continues for all blocks in the scoreboard.

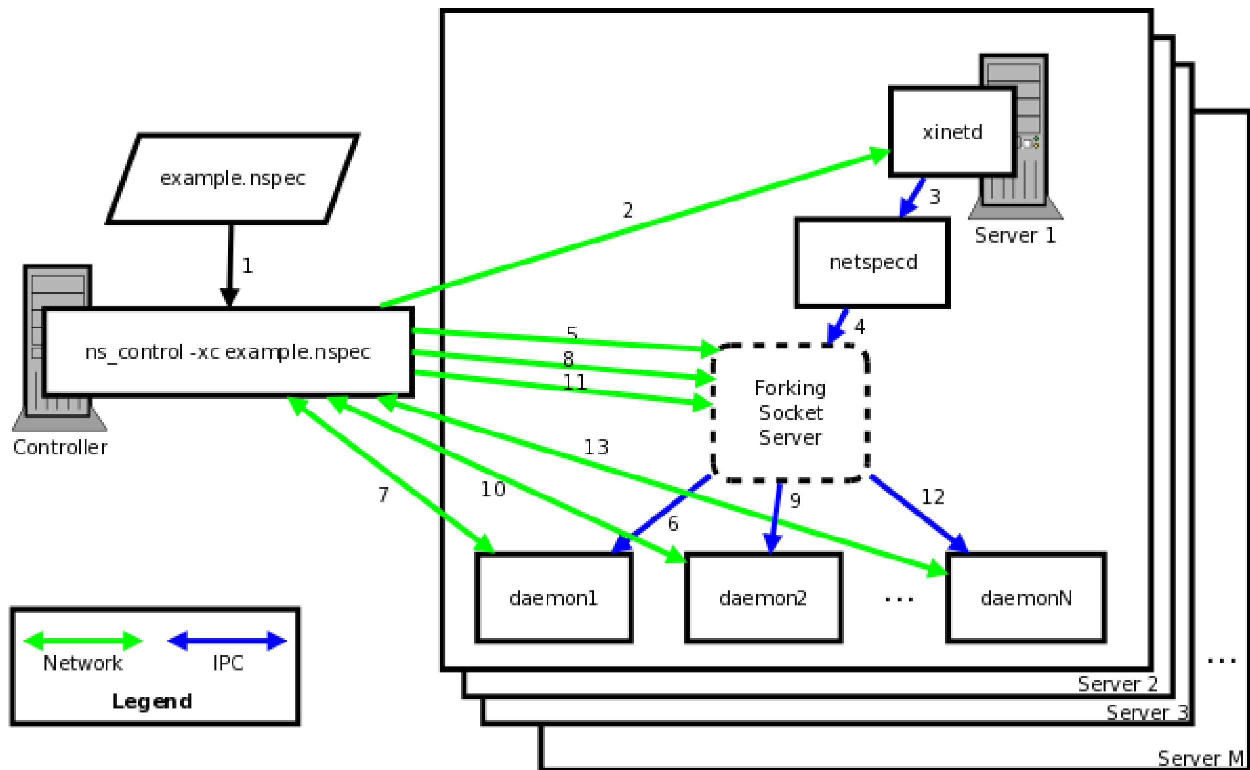

Figure 4: NetSpec Experimental Flow - Generalized

An example of a generalized NetSpec experiment is shown in Figure 4. The numbered steps are described as follows:

1) A NetSpec experiment starts with *ns_control* being launched from a machine that has network connectivity to all slave machines that will be involved in the experiment.

2) For the first machine specified in *example.nspec*, *ns_control* will make a socket connection to that machine's *xinetd/netspecd* port.

3) On the slave machine, *xinetd* will spawn the *netspecd* Python daemon.

4) *netspecd* will spawn a forking socket server that will listen for connections on the default NetSpec port.

5) *ns_control*, upon a successful connectino to *xinetd*, will create a new socket and connect to the default NetSpec port on the slave machine and send a packet containing the executable name of the first sub-daemon to spawn (*daemon1* in this example).

6) The forking socket server, having created a new process to handle the connection from *ns_control*, will for a new child process, *exec* the *daemon1* sub-daemon as a new process in said child, and hand context over to it.

7) The *daemon1* sub-damon will then communicate bi-directionally with *ns_control* using the socket that initiated the connection to *netspecd* (the network links shown for steps 5 and 7 are the same socket). From this point forward all phase start and completion messages are sent directly between *ns_control* and *daemon1* via this socket, **not** through *netspecd* and its forking socket server, as it only handles new connections.

8) See step 5, except the packet will specify *daemon2* as the sub-daemon executable.

9) See step 6, except *daemon2* will be *exec'*ed and given context.

10) See step 7, except *daemon2* will be communicating directly with *ns_control*.

11) See steps 5 and 8. The executable names for all remaining sub-daemons for this machine will be sent by *ns_control* using new socket connections to the default NetSpec port.

12) See steps 6 and 9. Each remaining sub-daemon will be *exec'*ed in a separate process.

13) See steps 7 and 10. Each remaining sub-daemon will communicate directly with *ns_control*.

The process described above for Server 1 is repeated for all remaining slave machines in the experiment (Server 2 through M in Figure 4). Once *ns_control* has received notification that all sub-daemons have been spawned and are waiting for phase messages, the experiment will officially begin. At this point *ns_control* sends phase start messages according to the scoreboard specified in the experiment control file.

A global timeline experiment requires coordinating the setup of data instrumentation and

execution of the distributed computation itself. Each of these can easily be specified as sub-daemons in a NetSpec control file. The entry for the DSKI daemon in an example control file is shown in Program 3. The name of the sub-daemon is *dski* and the executable is *dskictrl*. It is to be launched on *SERVER01, SERVER02,* and *SERVER03.* It declares one phase, *start*, which initializes the DSKI output file and enables the *NET_RX_ACTION_START* and *NET_RX_ACTION_END* events.

An example of a distributed computation sub-daemon entry is specified in Program 4. The name of the sub-daemon is *pipe1-01* and the executable is *netspec_sockets.py*. It is to be launched on SERVER01. It declares one phase, *setup*, which initializes several parameters controlling how the computation is to run on this server (the *netspec_sockets.py* program is discussed in detail in Chapter 4).

```
dski = {
    command = "dskictrl"
    host = [$(SERVER01), $(SERVER02), $(SERVER03)]
    workingdir = "$(WORKINGDIR)"
    phases = {
        start = {
            files = {}
            params = {
                dski = {
                    output_base = "$(DSUIFILEPREFIX).dski"
                    duration = 0
                    verbose = false
                }
                datastreams = {
                    dski_pipe = {
                        channel = default
                        filters = []
                        enabled = {
                            NET_DEVICE_LAYER = [
                                NET_RX_ACTION_START,
                                NET_RX_ACTION_END
                            ]
                        }
                    }
                }
            }
        }
    }
}
```

Program 3: NetSpec DSKI sub-daemon control file entry

```
pipe1-01 = {
    command = "./netspec_sockets.py"
    workingdir = "$(WORKINGDIR)"
    host = [$(SERVER01)]
    phases = {
        setup = {
            params = {
                role = "pipeline"
                protocol = "tcp"
                port = 50000
                buffer_size = 1024
                datafile = "/boot/vmlinuz"
                messages = 4000
                pipeline_length = $(PIPE1LENGTH)
                pipeline_rank = 1
                dsuifile = "$(DSUIFILEPREFIX).pipe1-01.bin"
            }
        }
    }
}
```

Program 4: NetSpec distributed computation sub-daemon control file entry

An example of a scoreboard schedule is shown in Program 5. The NetSpec controller, *ns_control,* will coordinate the experiment's execution by following this schedule. The experiment begins by running the *start* phase of the *dski* sub-daemon. This instantiates the DSKI subsystem on all servers, enables the specified events, and spawns off a process to collect events. The next step is to run the *setup* phase of all of the distributed computation sub-daemons (*pipe1-01, pipe1-02,* and *pipe1-03*). The sets up a series of socket pipelines between the servers over which random messages will be sent. Once setup is complete, the *sendrecv* phase is run, which starts sending messages over the pipelines. Note that this phase was not specified in the sub-daemon entry above. It is declared within the sub-daemon executable, but has no parameters. Thus it requires no phase configuration entry. After all messages have been sent, all sub-daemons are instructed to run the *cleanup* phase, which closes all connections and output files. The *ns_control* process will then accumulate all output files from every sub-daemon into one location. At this point the experiment is complete, and post-processing can be done on the output files to generate a global timeline visualization.

```
scoreboard = [
    {
        dski = "start"
    }
    {
        pipe1-01 = "setup"
        pipe1-02 = "setup"
        pipe1-03 = "setup"
    }
    {
        pipe1-01 = "sendrecv"
        pipe1-02 = "sendrecv"
        pipe1-03 = "sendrecv"
    }
    {
        pipe1-01 = "cleanup"
        pipe1-02 = "cleanup"
        pipe1-03 = "cleanup"
        dski = "cleanup"
    }
]
```

Program 5: NetSpec scoreboard schedule example

3.4 Post-Processing and Visualization

Before a global timeline visualization can be generated, all instrumentation events from the distributed experiment must be processed from their individual files into one large dataset. This process must perform several steps to ensure the final result is properly sorted. The first step is to sort events locally for each machine. For every machine, the DSKI and DSUI generate separate log files. Each of the events in these files include a system timestamp. As all of these timestamps are from the same server, it is guaranteed that a sort operation by time into one set will be correctly ordered.

Once a sorted dataset has been created for each server, the next step of sorting these into the complete global timeline dataset can proceed. Because the clock synchronization daemon was running on all servers during execution of the experiment, it is assumed that the event timestamps across all servers are accurate to within the offset reported by NTP. Thus the same sort-by-time operation is performed on all events. At this point all events have been sorted into a

single, global timeline dataset.

The final processing pass is to step through the dataset, converting each event into an X/Y plot point. The X-value of the point is the timestamp of the event. The Y-value is a constant depending upon which server the event was generated from. Each server in the computation is assigned two adjacent Y-values: one for its kernel events and another for its user events. Any events whose name match a set of predefined “event-pairs” will be saved. As the processing continues, if the matching pair event is found (by having the same unique ID stored in the extra data field, as discussed in Section 3.2), a vector is created between the plot points of both events. These plot points and vectors are output to a file in a format recognized by the GNUplot utility.

The DSUI includes a post-processing utility that can parse through the event log files and perform all of the steps discussed above. The behavior of this parsing is specified in a control file. The first section of the file, shown in Program 6, describes the per-server log file sort. This example is for an experiment with three servers.

```
@define RAWDATA ./gt_2server
@define RANK01 kusp32
@define RANK02 kusp33

    pipe01 = [
        head.input(
            file = [
                "$(RAWDATA)/$(RANK01)_pipe1-01/gt_2server.pipe1-01.bin"
                "$(RAWDATA)/$(RANK01)_dski/cpu0.bin"
            ]
        )
        utility.sort(
            sort_key = tsc
        )
        utility.timestamp(
            consume = false
        )
        utility.crop(
            start_event = "START/RANK1"
            end_event = "END/RANK1"
        )
    ]

    pipe02 = [
        head.input(
            file = [
```

```
                "$(RAWDATA)/$(RANK02)_pipe1-02/gt_2server.pipe1-02.bin",
                "$(RAWDATA)/$(RANK02)_dski/cpu0.bin"
            ]
        )
        utility.sort(
            sort_key = tsc
        )
        utility.timestamp(
            consume = false
        )
        utility.crop(
            start_event = "START/RANK2"
            end_event = "END/RANK2"
        )
    ]
```

Program 6: DSUI post-processing control file: per-server sort

The first block, denoted by *pipe01_dsui*, is a named dataset. This name will represent the finished dataset resulting from the operations in the control block, and can be referenced later in the control file. The first element in the block, *head.input*, denotes which files to use as input for processing. The files for this block are *kusp32_pipe1-02/gt_2server.pipe1-01.bin* and *kusp32_dski/cpu0.bin*, the DSUI and DSKI log files, respectively, from the *kusp32* server. The remaining elements specify the following:

- *utility.sort* – sort events by timestamp
- *utility.timestamp* – preserve the timestamp in the event, do not consume it
- *utility.crop* – crop the event stream using the specified start and end events

The second block specifies the same sort procedure on the files from the *kusp33* server. At this point there are two named datasets, *pipe01* and *pipe02,* that can be referenced by name later in the control file. The next section in the file is shown in Program 7. This details the parameters to pass to the custom global timeline filter (the DSUI postprocessing utility provides a framework for creating custom filters). This performs the final sort and generates the GNUplot datafile.

```
global = [
    head.input(
        conn = [pipe01, pipe02]
    )
    gt_filter.gnuplot(
        debug_level = 1
        datafile_prefix = "gt_2server"
        plot_filename = "gt_2server.plot"
        event_pairs = [
            { start_event = "SEND_MSG_RANK1"
              end_event = "RECV_MSG_RANK2"
              threshold = 0 },
            { start_event = "SEND_MSG_RANK2"
              end_event = "RECV_MSG_RANK1" }
        ]
        pair_datafield = "extra"
        unknown_pairs = true
    )
]
```

Program 7: DSUI post-processing control file: global timeline filter

The *head.input* section instructs the utility to sort the two per-server datasets (*pipe01* and *pipe02*). This will sort all events into one dataset by timestamp. The next block, *gt_filter.gnuplot*, will parse through this final dataset and generate GNUplot points and vectors. Most of the parameters to the filter are self-explanatory. The *event_pairs*, *pair_datafield*, and *unknown_pairs* parameters require discussion and are explained as follows:

- *event_pairs* – The list of event name pairs and associated floating point thresholds to use when drawing vectors between events. Each pair of events that share the same unique ID, whose event names match one of these entries in the correct order and whose time difference is less than the specified threshold will be drawn as a vector. Those that exceed the threshold will still be rendered as vectors, but will also print a warning message. If no threshold parameter is specified, it will default to 0, which is the same as an unlimited threshold.
- *pair_datafield* – The event data field to use when searching for pairwise unique identifiers.

- *unknown_pairs* – If set to *true* and any unknown pairs are found (i.e. their event names do not match the entries specified in *event_pairs*), print a warning message and render vectors for them. Ignore them if set to false (the default value).

GNUplot visualizations resulting from control files similar to the above example are presented and discussed in Chapter 4.

4 Evaluation

A thorough evaluation of the proposed global timeline results is presented in two parts: clock synchronization and visualization. Clock synchronization must be discussed first, as it should be shown to be working to a reasonable extent. Otherwise the timestamps used to sort the instrumentation datasets will be erroneous and the resulting GNUplot visualizations will be invalid.

4.1 Clock Synchronization

To verify that our NTP modifications are synchronizing clocks to within microsecond resolution, there are two usable benchmarks: the offsets reported by the NTP client and a regression test that measures message transmit times between servers using timestamps from both endpoints. The first benchmark is implemented as a daemon that runs in the background during all global timeline experiments. The second can be run outside of experimentation periods as a sanity-check on the clock synchronization subsystem.

4.1.1 Clock Synchronization Daemon

Each server in a global timeline experiment must call a modified version of the NTP *ntpdate* utility, which will request time offset measurements from a server running a modified version of the NTP *ntpd* daemon. These modifications, as discussed previously, utilize finer-grained timestamps taken from within the kernel to achieve better resolution. The *ntpdate* utility can be called in one-shot fashion, performing a one-time offset. This is not ideal for an experiment, especially lengthy ones, as clocks on COTS systems will always drift over time and

it is impossible to find a perfect synchronization offset. The standard NTP client daemon that runs on most Linux systems routinely performs offset measurements and clock adjustments. We created a similar daemon, called *kusp_clksync*, to call our modified NTP *ntpdate* repeatedly to fine-tune the clock. This daemon runs on all servers in the global timeline experiment.

```
[dinkel@kusp33 ~]$ tail -f /var/log/kusp_clksync.log
Sync thread started
initial adjustment -29031.18
raw offset (us): 0.031   smoothed offset (us): 0.031
We seem to be in sync.
Now sleeping for 10 seconds.
raw offset (us): 0.795   smoothed offset (us): 0.795
Now sleeping for 15 seconds.
```

Program 8: Clock synchronization daemon output

Example output from the *kusp_clksync* log file is shown in Program 8. After the synchronization thread is started, the modified *ntpdate* is called to accumulate a set of time offsets. No adjustments are made during this initial period. Once a predetermined number of offsets have been measured, and average offset is calculated along with an accompanying clock adjustment. This is the first adjustment made to the system clock (reported as -29031.18 microseconds in this example). After this adjustment, another *ntpdate* call is made to measure the resulting time offset, shown as 0.031 microseconds. As this is less than 10 microseconds (a default but configurable threshold for our modified *ntpdate*), synchronization is assumed.

At this point the daemon will sleep for per-determined periods of time, wakening to re-run *ntpdate* to verify the offset. If the offset is still within the threshold, the sleep time will be extended. This example shows an initial sleep of 10 seconds, followed by an observed offset of 0.795 microseconds, and then an extended sleep of 15 seconds. As long as offsets are within the threshold, no adjustments are made. This keeps adjustments to a minimum.

This same process is repeated in the background on all servers in the global timeline experiment. Each server communicates with the same time server running our modified *ntpd*

master daemon, ensuring that all clocks are synchronized to the same source.

4.1.2 Regression Test

To further verify the functionality of our clock synchronization subsystem, a regression test is run to compare clock timestamps between servers. This test is a simple user-mode application that measures the packet transmission time between two servers that have already been synchronized to the same NTP time server. Timestamps are taken immediately before send calls on one server and immediately after receive calls on the other server. Assuming clocks have been synchronized, a distribution of the difference of these timestamps (i.e. the transmission time) should be tightly packed and with an average transmission time on the order of 10s of microseconds [22].

As discussed in Section 3.1, the use of interrupt coalescing in the e1000 driver can introduce error into the clock synchronization measurements. Figures 5 and 6 illustrate this behavior, where the majority of the transmission times are not tightly packed and range into the low 100s of microseconds. This is due to there being no guarantee that the network message will be delivered up the stack to the user-level regression test immediately after its related device interrupt. The timestamp retrieval is at the mercy of the e1000 device driver's determination that enough interrupts have been received to warrant processing of the message queue.

With interrupt coalescing disabled, the transmission times fall more in line with expectations, as shown in Figures 7 and 8. Distribution of the times in both directions is tightly packed, with a spread of roughly 10 microseconds.

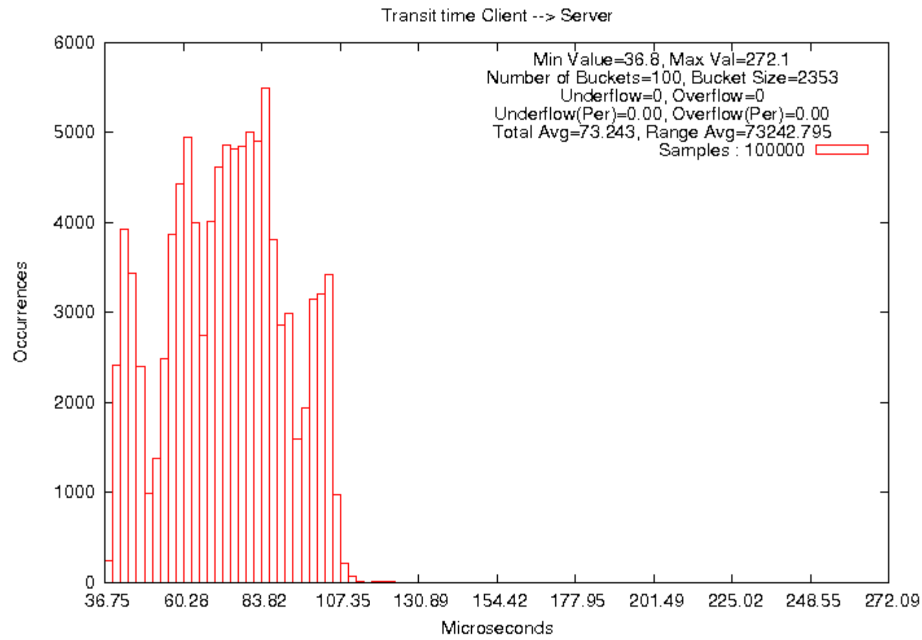

Figure 5: Clock synchronization regression test - NAPI on (client to server)

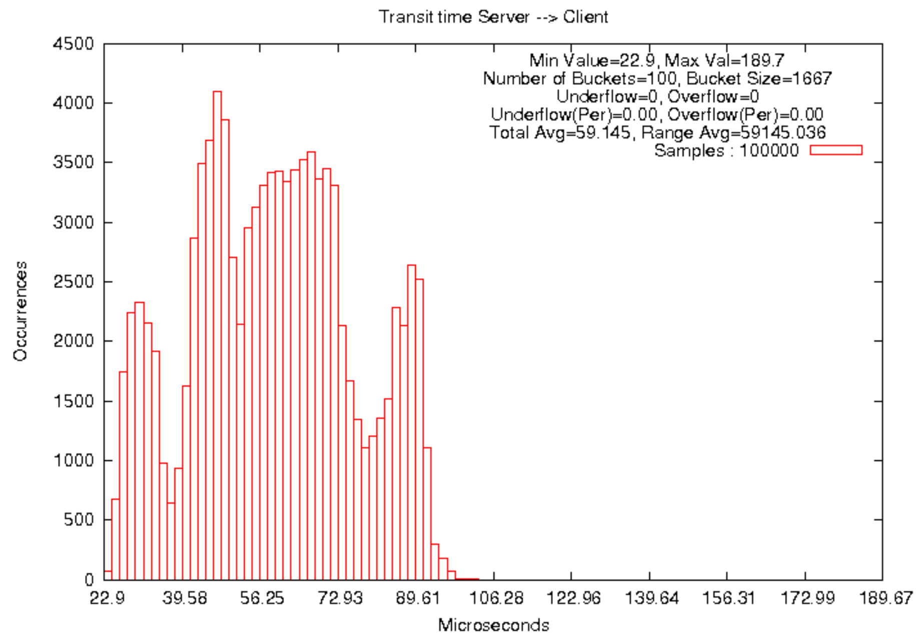

Figure 6: Clock synchronization regression test - NAPI on (server to client)

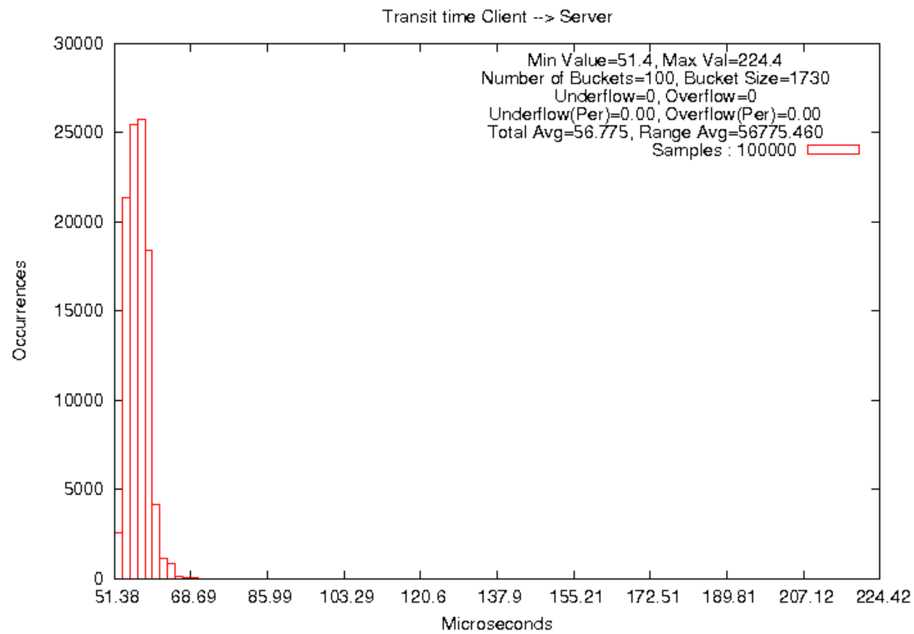

Figure 7: Clock synchronization regression test - NAPI off (client to server)

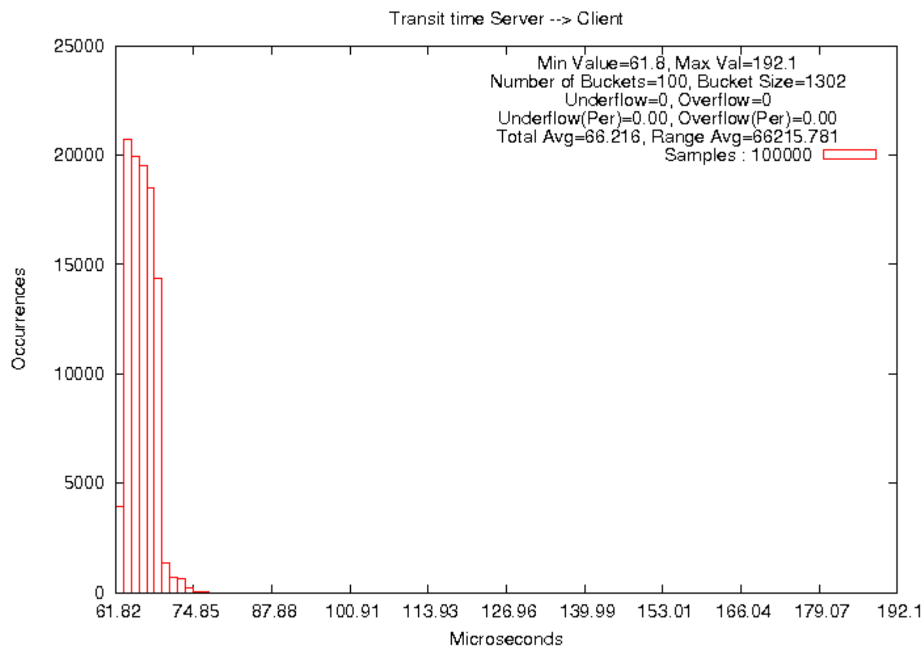

Figure 8: Clock synchronization regression test - NAPI off (server to client)

4.2 Post-Processing and Visualization

The example distributed computation chosen for evaluation of our global timeline approach is a simple socket pipeline between three servers, executed by a NetSpec sub-daemon called *netspec_sockets.py*. A file, broken into equal-length messages, is sent through the pipeline from the first server, passing through the second, and ending at the third, where it is reassembled into a copy of the file. Each server logs DSUI events when messages are sent and received. These same send and receive events are grouped into pairs during postprocessing, allowing for vectors to be drawn between them in the visualization. The DSKI *NET_RX_ACTION_START* and *NET_RX_ACTION_END* events, referenced in Section 3.3, are also generated on each machine. These cannot easily be "connected" across machine boundaries with extra pair-wise data, but their presence as events in the global timeline provide a useful illustration of the relationship between kernel and user-level activity.

The experiment is coordinated via NetSpec, using the control file presented in Section 3.3. A graphical representation of the experimental flow is shown in Figure 9. Steps 1 through 7 were explained previously in Section 3.3, under the NetSpec generalized example. Steps 8 through 10 signify the network socket pairs between each server that make up the pipeline. Data will flow from *kusp32* to *kusp33* and finally to *kusp34.*

Upon completion of the NetSpec run, the server where *ns_control* was launched will have a set of Data Streams output files from every machine in the experiment. These are what will be processed into a global timeline. The post-processing control file is similar to that presented in Section 3.4, with the exception of the global timeline filter block specifying several more event pairs in the *event_pairs* entry (see Program 9). This will create vectors in the global timeline visualization from *kusp32* to *kusp33*, as well as from *kusp33* to *kusp34*.

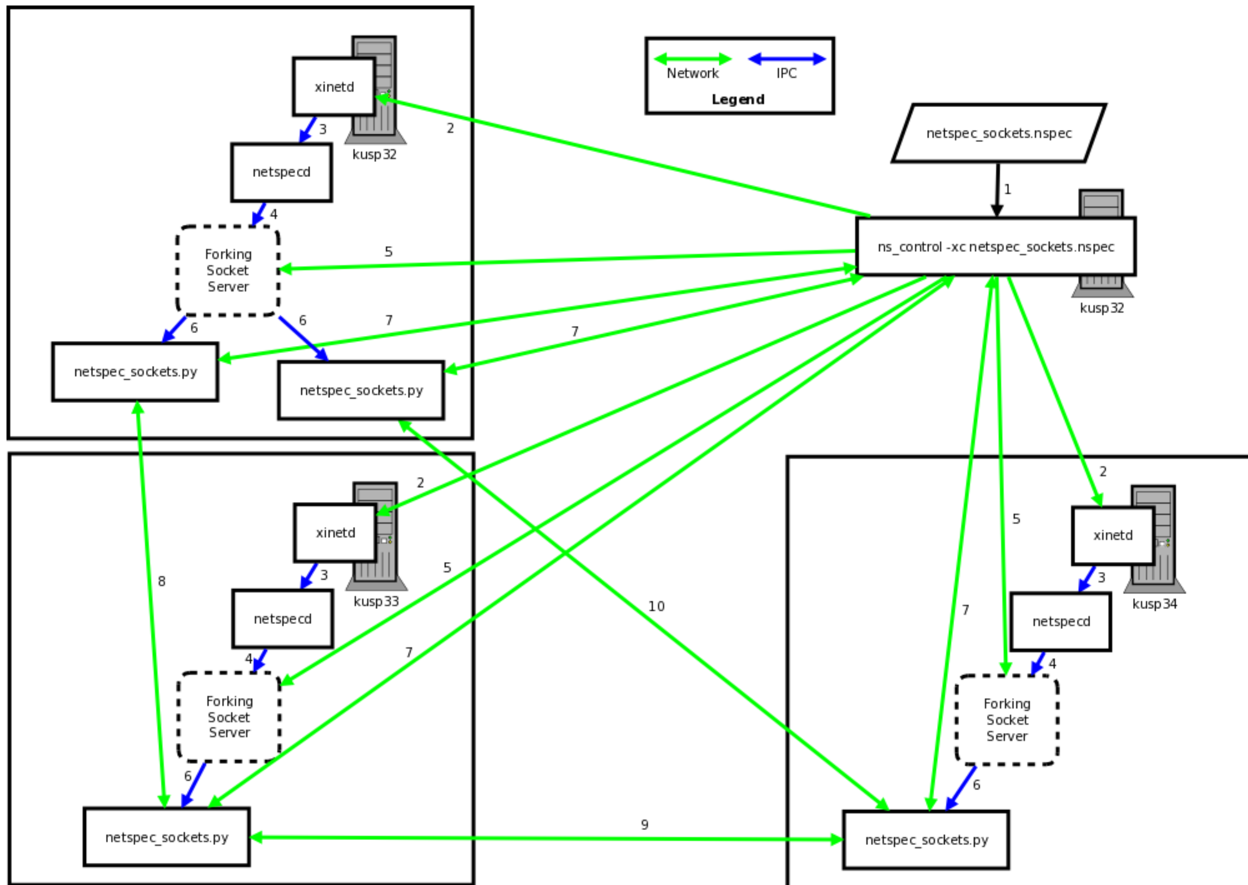

Figure 9: NetSpec Experiment Flow - 3 server socket pipeline

```
event_pairs = [
      { start_event = "SEND_MSG_RANK1"
        end_event = "RECV_MSG_RANK2" },
      { start_event = "SEND_MSG_RANK2"
        end_event = "RECV_MSG_RANK1" }
      { start_event = "SEND_MSG_RANK2"
        end_event = "RECV_MSG_RANK3" },
      { start_event = "SEND_MSG_RANK3"
        end_event = "RECV_MSG_RANK2" }
 ]
```

Program 9: Event pairs - 3 server socket pipeline

Launching the post-processing utility on this control file results in the console output shown in Program 10. Note how all events are processed twice: once to sort via timestamp into a single dataset, and the second time to generate the corresponding GNUplot points and vectors.

```
[dinkel@kusp32 tmp]$ postprocess -f kusp/share/gt_3server.pipes
Executing Pipeline Configuration `kusp/share/gt_3server.pipes'
Event pairs:  [{'threshold': 0.0, 'start_event': 'SEND_MSG_RANK1',
'end_event': 'RECV_MSG_RANK2'}, {'threshold': 0.0, 'start_event':
'SEND_MSG_RANK2', 'end_event': 'RECV_MSG_RANK1'}, {'threshold': 0.0,
'start_event': 'SEND_MSG_RANK2', 'end_event': 'RECV_MSG_RANK3'},
{'threshold': 0.0, 'start_event': 'SEND_MSG_RANK3', 'end_event':
'RECV_MSG_RANK2'}]
Processing global timeline events...
Processed 10000 events.
Processed 20000 events.
Processed 30000 events.
Processed 40000 events.
Processed 50000 events.
Processed 60000 events.
Processed 70000 events.
Writing gnuplot data files...
Processed 10000 events.
Processed 20000 events.
Processed 30000 events.
Processed 40000 events.
Processed 50000 events.
Processed 60000 events.
Processed 70000 events.
WARNING: Creating a data file for the unknown pair id 072770000003368
( ['kusp32', 2149312.5, 'RECV_MSG_RANK1', 'user', 'RANK1'] to ['kusp33',
2149325.75, 'SEND_MSG_RANK2', 'user', 'RANK2'] )

Gnuplot plot file gt_3server.plot has been generated.
Postprocessing complete.
```

Program 10: Global timeline post-processing console output

Running GNUplot on the generated script file produces the graph shown in Figure 10. As this is the entire experiment from start to end, all of the more than 70000 events are shown. This time scale is far to great to show any detail, with all events packed together into seamless blocks of color. The GNUplot utility provides interactive zooming, though, and after several progressive zoom operations into the section where DSUI events start occurring, a much more informational view appears.

Referencing Figure 11, several conclusions can be drawn from this global timeline visualization:

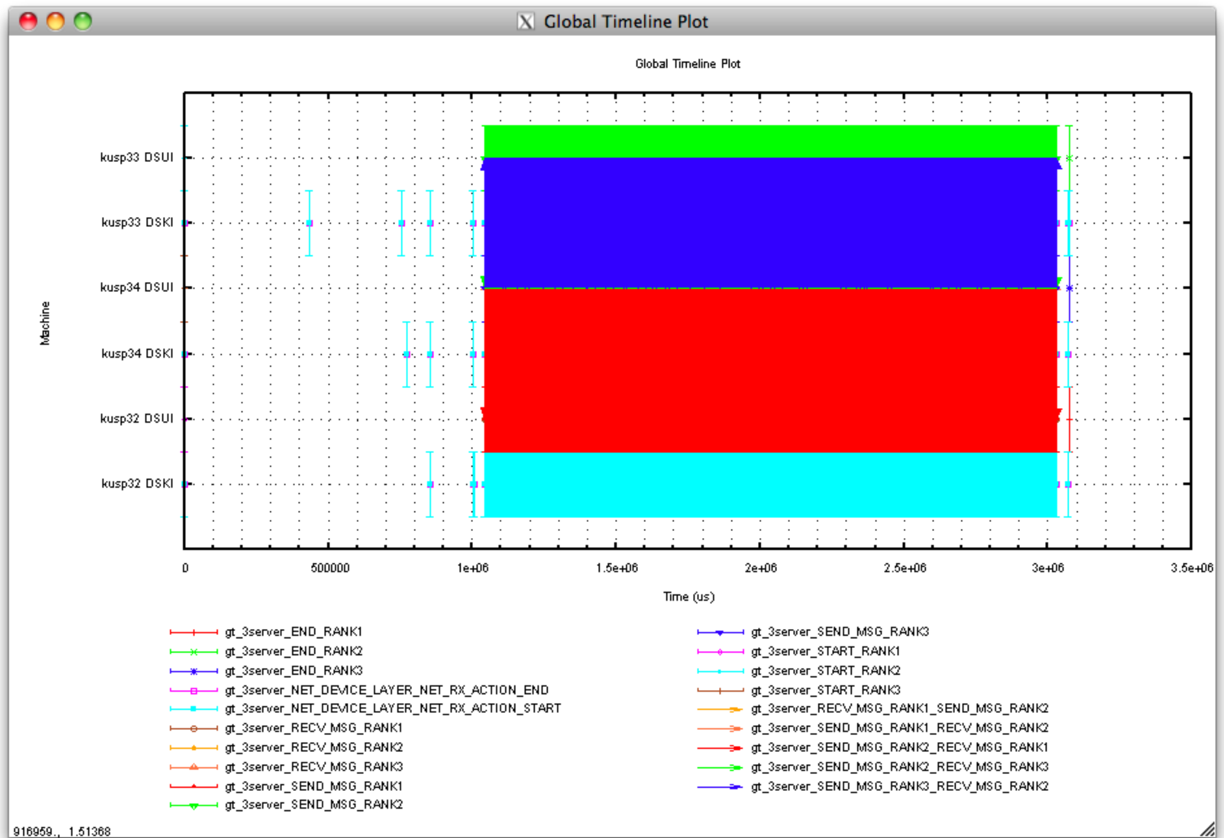

Figure 10: Global timeline plot - Full

- The DSKI soft-interrupt start and end events occur just before a DSUI receive event in the pipeline computation is logged on the same server. Other unrelated soft-interrupt events appear as well, which are most likely associated with packet arrivals for unrelated system processes. See Chapter 5 for a discussion of how these could be filtered.
- All DSUI send and receive events are connected with vectors.
- All DSUI send events take place earlier than their corresponding receive events. This is further validation that the clock synchronization process is working, as the timestamps used to sort these events came from *different servers*.

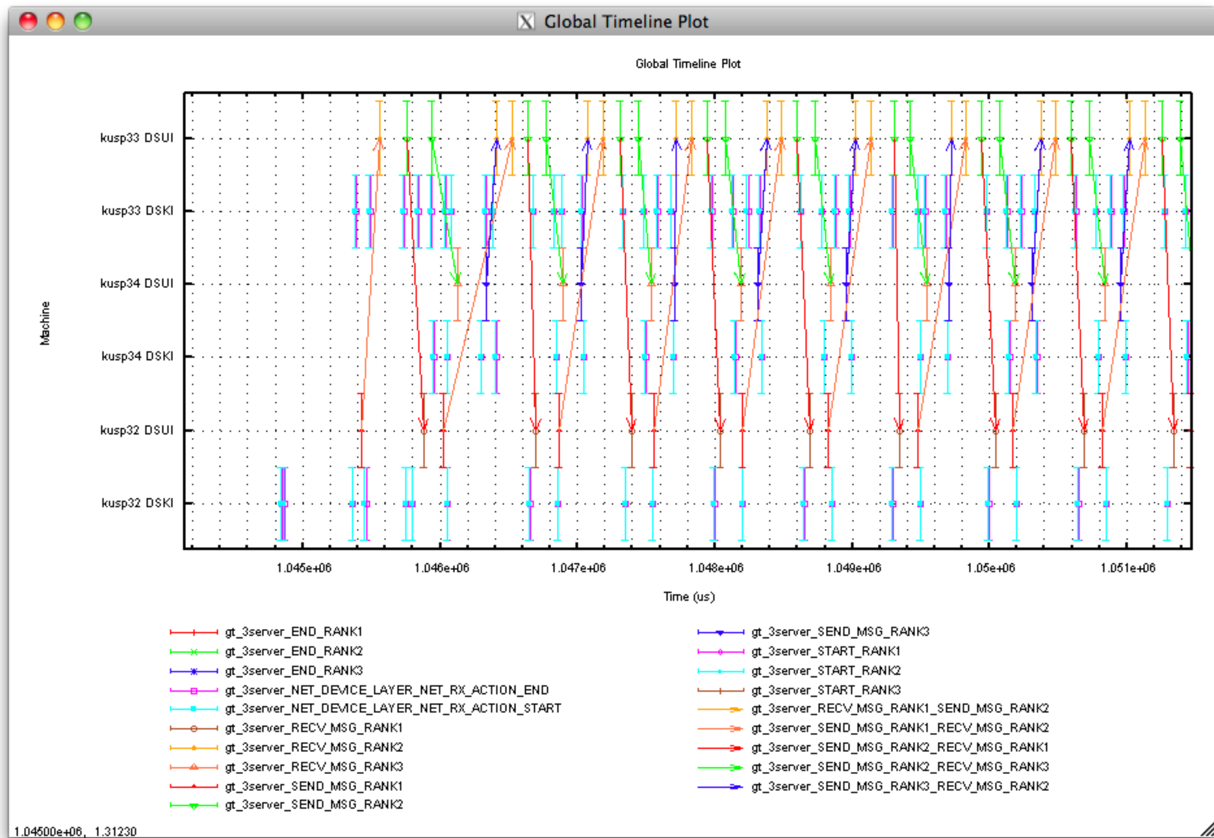

Figure 11: Global timeline plot - DSUI event start

Of particular interest is the warning message printed in the post-processing output in Program 10. This indicates an out-of-order event sequence from *RECV_MSG_RANK1* to *SEND_MSG_RANK2* beginning at time 2149312.5. The receive event timestamp was found to be earlier that that of the send event. Zooming into this area of the global timeline plot produces the image in Figure 12, with the sequence in question highlighted by the red rectangle. The *SEND_MSG_RANK2* event on *kusp33* should occur before the *NET_RX_ACTION* DSKI event sequence on *kusp32*, but it was somehow delayed until after *kusp32's RECV_MSG_RANK1* event was logged. A possible explanation for this is that the *netspec_sockets.py* process on *kusp33* had context taken away from it immediately after sending its message but *before* it could log the

corresponding DSUI send event. Repeating the experiment with additional DSKI events enabled, such as the context switch events, could shed light on this scenario, provided similar out-of-order events occur in future runs.

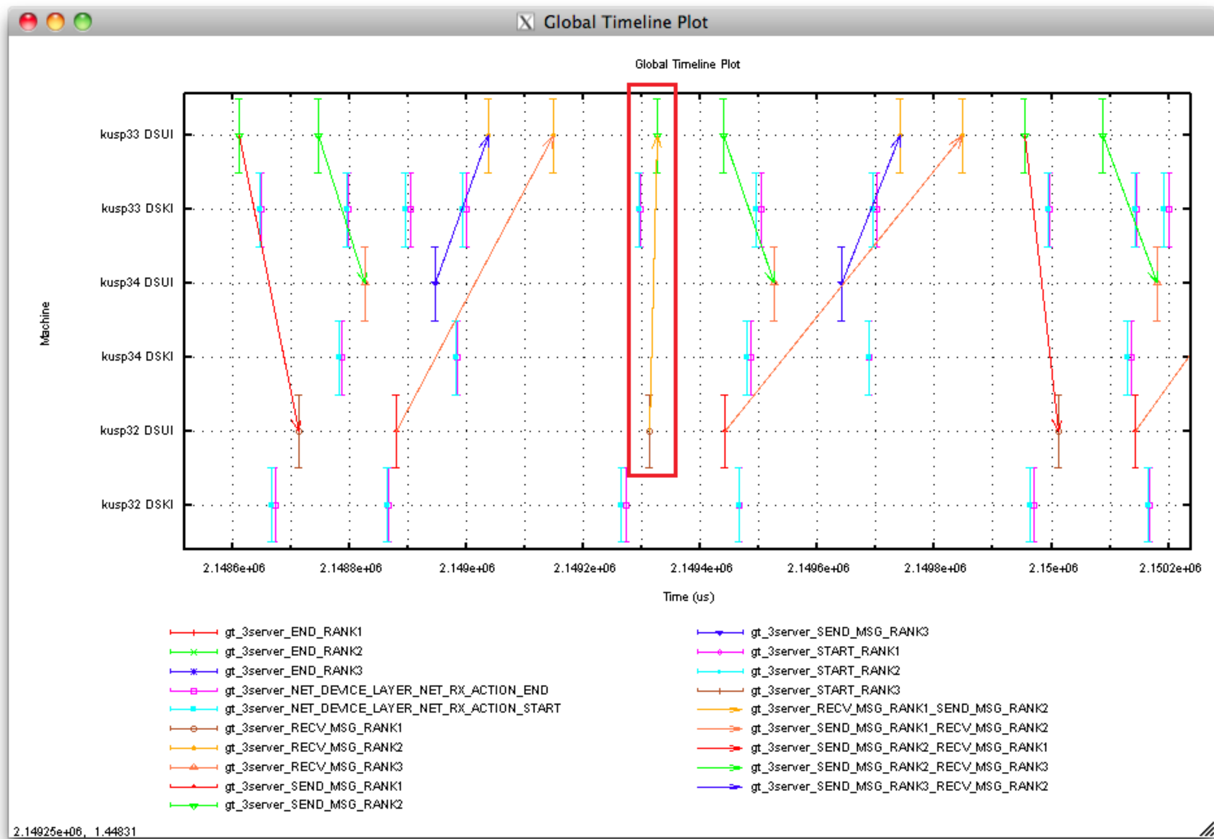

Figure 12: Global timeline plot - out-of-order event sequence

5 Conclusions and Future Work

With data centers expanding to accommodate more servers [23], it is evident that distributed computing is on the rise. To justify this expansion, evaluation of the computations running on these servers is required to ensure that these resources are fully utilized. We have proposed a global timeline visualization of a distributed computation as a method for performing this evaluation. This was built on COTS servers running the Linux operating system. Instrumentation events from both the kernel and user-level were gathered. The timestamps for these events were accurate to within microseconds by synchronizing all server clocks with a modified NTP time service. The resulting event stream data was converted to point and vector types readable by the GNUplot utility and rendered on a plot. The resulting plots were used to observe flow of the computation across servers, and were helpful in highlighting a potential execution anomaly warranting further evaluation.

This further evaluation could involve additional instrumentation points in the kernel, such as scheduler context switch events. The drawback is that the volume of these would quickly become unwieldy, requiring run-time filtering to report only the switch events for the user processes involved in the distributed computation. This “active filtering” is part of ongoing research in the DSKI and DSUI.

Our NTP subsystem modifications required that the network device driver and kernel network stack be modified as well. This is a less-than-ideal solution in that it introduces hardware dependency. In the time since this research was conducted, the Linux PTP implementation has become a viable alternative for high-resolution clock synchronization. Replacing our modified NTP subsystem with PTP would introduce support for many more

network interfaces and processor architectures and potentially eliminate kernel modifications. Investigation is required to measure the overhead of any PTP timestamp retrieval calls, as this is an important factor when logging instrumentation events, especially in the kernel. Both the DSKI and DSUI require lightweight timestamping routines in order to prevent the adverse effect of instrumentation introducing too much overhead into the computation.

Visualization could be expanded to draw vectors between related kernel network events, such as packet enqueue on the sending server and dequeue on the receiving side. A potential unique id to match these events could be found in the TCP sequence number. Initial attempts at this solution resulted in unmatched pairs, and further investigation is required.

Another visualization improvement would be to provide more interactive plot features, such as enabling/disabling the display of selected event types as well as scrolling the timeline horizontally. The GNUplot utility does not provide an API for custom interaction. Porting the global timeline post-processing filter to export data usable by other graphing packages, such as *matplotlib*, would allow for more customization.

6 References

[1] Coulouris, George; Dollimore, Jean; Kindberg, Tim; Blair, Gordon. Distributed Systems: Concepts and Design. 2012.

[2] Massie, Matthew L.; Chun, Brent N.; Culler, David E.. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Parallel Computing, vol. 30, issue 7, July 2004.

[3] "List Statistics | Top500 Supercomputer Sites". http://www.top500.org/statistics/list/. Retrieved 18 June 2013.

[4] "NTP FAQ: How does it work?". http://www.ntp.org/ntpfaq/NTP-s-algo.htm. Retrieved 24 June 2013.

[5] Rostedt, Steven. "Debugging the kernel using Ftrace". 2009. http://www.ntp.org/ntpfaq/NTP-s-algo.htm

[6] Rentel, C.H.; Kunz, T.. A clock-sampling mutual network time-synchronization algorithm for wireless ad hoc networks. In Proceedings IEEE Wireless Communications and Networking Conference, 2005.

[7] McKnight-MacNeil, Ereth. CS-MNS: Analysis and Implementation. Ottawa-Carleton Institute for Electrical and Computer Engineering, 2010.

[8] "IEEE-1588 Standard Version 2 - A Tutorial". http://www.webcitation.org/5qaJpYqCH. Retrieved 18 June 2010.

[9] Ohly, Patrick; Lombard, David N.; Stanton, Kevin B.. Hardware Assisted Precision Time Protocol. Design and case study.. In Proceedings Linux Cluster Institute, 2008.

[10] "The Linux PTP Project". http://linuxptp.sourceforge.net/. Retrieved 24 June 2013.

[11] Eigler, Frank Ch.. Systemtap tutorial. 2013. http://sourceware.org/systemtap/tutorial/tutorial.html

[12] Eigler, Frank Ch.. Problem Solving With Systemtap. Red Hat, 2006.

[13] Fournier, P-M; Desnoyers, M; Dagenais MR. Combined Tracing of the Kernel and Applications with LTTng. In Proceedings Linux Symposium, 2009.

[14] Sundaresan, Anupama. NetSpec User Manual. ITTC, University of Kansas, 1999.

[15] Gabriel, Edgar; Fagg, Graham E.; Bosilca, George; Angskun, Thara; et. al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In Proceedings European PVM/MPI Users' Group Conference, 2004.

[16] Linux Trace Toolkit Viewer User Guide. http://lttng.org/files/lttv-doc/user_guide/. Retrieved 25 June 2013.

[17] Zaki, Omer; Lusk, Ewing; Gropp, William; Swider, Deborah. Toward Scalable Performance Visualization with Jumpshot. Mathematics and Computer Science Division, Argonne National Laboratory, 1999.

[18] The Network Time Protocol (NTP) Distribution. http://www.eecis.udel.edu/~mills/ntp/html/index.html. Retrieved 25 June 2013.

[19] Mills, David L.. Computer Network Time Synchronization: the Network Time Protocol on Earth and in Space. 2011.

[20] Subramanian, Hariharan. Systems Performance Evaluation Methods for Distributed Systems Using Data Streams. University of Kansas Electrical Engineering and Computer Science Department, 2005.

[21] Buchanan, Brian R.; Niehaus, Douglas; Dhandapani, Gowri; Menon, Raghavan; et. al. The Data Stream Kernel Interface. ITTC, University of Kansas, 1998.

[22] Larsen, Steen; Sarangam, Parthasarathy; Huggahalli, Ram; Kulkarni, Siddharth.

Architectural breakdown of end-to-end latency in a TCP/IP network. Int. J. Parallel Program., 2009

[23] Novet, Jordan. Google eyes two-story Oregon data center design to maximize efficiency. 2013. http://gigaom.com/2013/06/10/google-eyes-two-story-oregon-data-center-design-to-maximize-efficiency/