Application-Kernel Collaboration Mechanisms for Real-Time Cluster Server under Overloading

Yu Tang[#], Rahul Simha[#], Shuyu Chen[*], Changqing Bu[*], Guanghui Chang[*]
[#]*Department of Computer Science, The George Washington University*
Washington, DC 20052, U. S. A.
{yutang, simha}@gwu.edu
[*]*School of Software Engineering, Chongqing University*
Chongqing 400030, P.R. China
sychen@cqu.edu.cn

Abstract

Cluster-based servers delivering timely responsive service can shorten response latency and maximize system throughput through multithreading. However, under high workload, large volume of threads may overload the kernel, leading to an inoperational system "hold-out" status. Majority of overload control work have been done at application level, but lack the collaboration between application and kernel to proactively respond to overloading. In this paper we propose two application-kernel cooperative mechanisms, of which the Flush-Out function recovers system from overloading by filtering out certain amount of events from kernel, and the Early-Drop mechanism protects system from overloading by proactively responding to load status. Experiments on a cluster server indicate the proposed mechanisms improve server's responsiveness under high load condition by substantially cutting the response time by 7~22% and event drop rate to 10~21%. The application-kernel mechanisms demonstrate its effectiveness in keeping mission-critical servers in operational state and delivering improved performance under high workload.

1. Introduction

Cluster-based servers built by using commodity processors and COTS high-speed networks have been considered a technically efficient and cost-effective computing platform for the applications operated in critical environment to deliver highly available and timely responsive service. These applications include battlefield surveillance sensor networks [1, 2], environmental monitoring systems [3, 4], and infrastructure protection and health sensing systems [5], in which the application server is demanded to support a large volume of distributed sensor nodes with varied performance requirements. Typically, the client's service requests (called events or messages) by its nature can be categorized into two classes: real-time or non-real-time. The real-time events come in with critical timely constraints, and any delay in response to these events may lead to unacceptable or even disastrous consequences. One example is US Air Force's Sensor-to-Shooter tactics in precision strike, which requires the targeting circle time to be less than 10 minutes [6]. The events reading routine environmental data or monitoring non-critical part of structures fall in the non-real-time category, of which the performance goal is to maximize server throughput so more sensor events gets processed.

In order to meet the two-dimensional performance goals imposed by two classes of events, the server tends to be programmed in multithreading mode, so it can benefit from concurrency and take advantage of multi-core CPUs or multi-processor systems. A common practice of multithreading is the thread pool model [7], in which a limited number of threads forms a thread pool and a thread is selected from the pool to service the client request when it arrives. However, this approach is not suitable for the applications bound to response latency, since waiting for available thread in the pool adds to response latency. In our application, a service thread is spawn immediately and exclusively for each event so that it can be serviced without any delay. Linux uses a one-on-one thread approach [8], in which thread is scheduled as a kernel task in kernel space. This raises an issue that an extremely high volume of sensor events may generate hundreds or thousands of threads in kernel instantly, which can overload the kernel and lead to an inoperational system "hold-out" status. Overloading control and recovery has become a critical issue to the application servers operating under high workload.

Majority of overload control work has been done at the application layer by implementing certain type of request admission mechanism that only starts service when load level is below server's capacity [9, 10]. To the real-time cluster server (RTCS) [11], this approach adds complexity to server implementation and incurs undesired application level buffering delay. Voigt et al. [12] proposed a set of kernel-based overload control and service differentiation mechanisms in which the

acceptance of incoming requests is based on the connection's attributes or priorities, and the packets are discarded if the allocated rate is exceeded. This approach effectively regulates the rate and burst level of connections, however, does not provide capability of recovering the system if overloading ever happens. Also the header parsing and classification is timely inefficient to the applications bound to response delay.

Based on above observations, we propose two application-kernel mechanisms that target on

- providing a mechanism to recover the kernel from the hold-out state when it is overloaded;
- provide a mechanism to proactively respond to workload level and avoid kernel overloading;
- provide a tuning knob to allow the application to select one performance metrics over another.

The rest of paper is organized as follows: Section 2 describes the cluster server's architecture and multithreading model; Section 3 presents the design of kernel Flush-Out and Early-Drop mechanisms; Section 4 conducts a series of kernel experiments and analyzes the results. In Section 5 we summarize our findings in experiments and the issues remained for future work.

2. Server multithreading model

The Real-Time Cluster Server (RTCS) system [11] consists of a pool of computing nodes (PC or workstation) interconnected through high-speed LAN such as Gigabit Ethernet [14]. Configured with the NAT (Network Address Translation) configuration, the cluster server has a front-end node as the access point to external sensor network. The distributed remote sensors exchange messages (events) with the cluster server via the front-end node. Within the cluster, the arriving events are dispatched to backend nodes for service under certain cluster scheduling schemes. At each backend node, there is an application server running to service the sensor events.

The backend application server implements three functional threads to perform the tasks, i.e., the Read_Evt thread, the Sche_Evt thread, and the service thread, shown in Figure 1. The Read_Evt thread reads the events off the network interface and classifies them into two events queues. At the other end of the queues, the Sche_Evt thread selects an event out of the queues under the Queue Length Proportional (QLP) algorithm [13], and spawns a service thread to serve the event.

The application server can execute above steps in two different modes, i.e. the sequential mode or the multithreaded mode. Under sequential mode, the server services the event in a one-by-one style, in which the Sche_Evt thread will not start a new service thread until the previous one is completed. Under multithreaded mode, rather than wait for previous service thread returns, the Sche_Evt thread keeps fetching events from the queue and immediately starting a new thread for the event. Under such a mode, the server can service multiple events simultaneously by having multiple service threads concurrently running in the kernel.

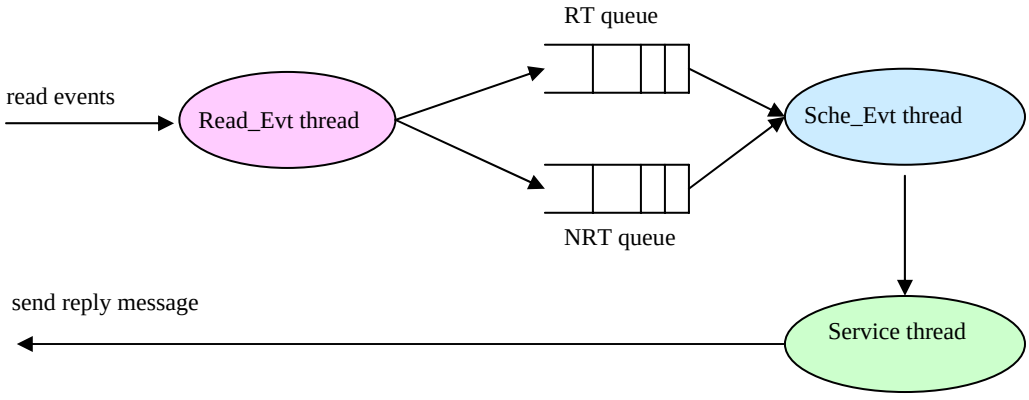


Figure 1. Threads of backend server

3. Application-Kernel collaboration

In our approach, two kernel mechanisms, namely Early-Drop and Flush-Out, are implemented to address the kernel overloading issue. The Early-Drop mechanism is used to dismiss sensor events early in kernel space before handed up to the upper application. The Flush-Out function works as a rescue operation to recover the system from hold-out status by filtering out certain amount of event tasks.

3.1 Kernel Early-Drop

The Early-Drop mechanism consists of two parts: application control and kernel drop action. The application control simply uses the number of active kernel tasks (retrieved by system call *get_nr_running()*) as the yardstick to measure kernel's load status, and starts/stops kernel drop action by setting/unsetting an EARLY_DROP flag, alternatively. The drop action is performed by the kernel scheduler, which keeps check

the EARLY_DROP flag. If finds the flag set to TRUE, the scheduler makes a call to the socket module to dump the arriving event packets rather than copy them to upper layer for processing; if FALSE, it stops the dropping action and returns to normal operation.

Both [9] and [12] mentioned the useless TCP packet retransmission issue caused by dropping TCP connections in the transport layer. To deal with this issue and avoid interruption to upper application, we add a switch to the UNIX system call *read()* to change the data copying sequence between the kernel and application layer. The normal *read()* calls a kernel function *sys_read()* to load the event data into the application's buffer for further processing, shown as the blue path in Figure 2. In our solution, a new kernel function *rt_sys_read()* is added to kernel and let *read()* call this new function rather than original *sys_read()*, shown as the red path in Figure 2. Within the new *rt_sys_read()* call, it always checks the EARLY_DROP flag first, and decides whether to proceed to original *sys_read()* call to copy the events to application buffer, or to dump the events by returning an empty buffer. Since this change is implemented post to the reassembly of the TCP packets in the transport layer, it would not trigger undesired TCP retransmission caused by the improperly dropping of SYN packets in transport layer.

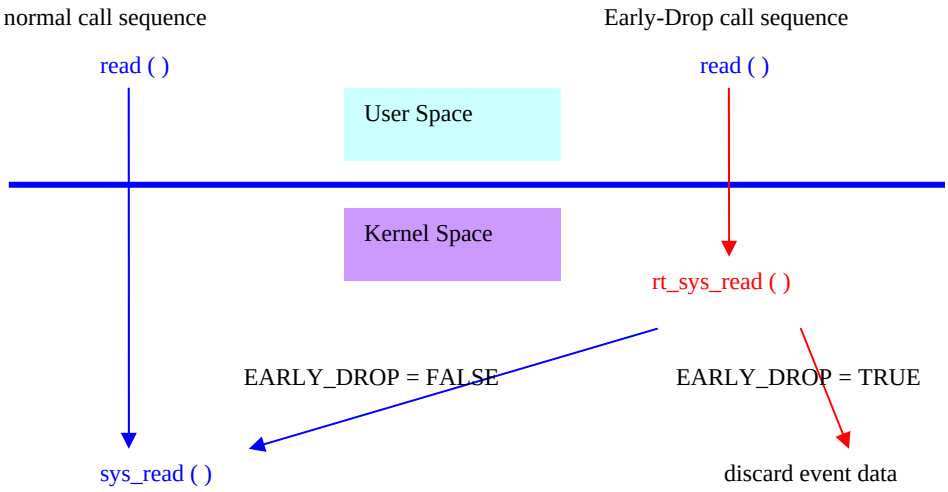


Figure 2. Early-Drop call sequence

3.2 Kernel Flush-Out

When pulled into "hold-out" state due to kernel overloading, the system shall be capable of recovering by flushing out certain amount of events from kernel space, or more accurately, flushing out kernel tasks mapped to certain class of events. Linux provides system calls to kill a running process or thread, yet two issues make this approach inappropriate or inefficient to delivering timely responsive service.

First, our operation requires remove kernel tasks by the sensor event attributes such as sensor ID or priority level, but Linux killing calls use process ID or thread handle as the arguments. Thus the application has to maintain a look-up table to map the event attributes to the process ID or thread handle, which is very inefficient. Second, even the process ID or thread handle are sorted out, the application is still not aware of the status of the process or thread in kernel until it makes extra calls to retrieve process/thread state. Trying to kill a zombie process or thread is not only a waste of resource, but may lead to unpredicted behavior as well. Apparently, embedding Flush-Out function to kernel would be more efficient, and relieves the application from the process/thread mapping and look-up overhead.

In Linux kernel, each thread is treated as a regular task identified by a structure *task_struct*, and scheduled by the kernel scheduler. In order to identify the kernel tasks by the event's attributes, we add two new fields, *sensor_tag* and *event_mark*, to this task structure:

```
struct task_struct {
        ……
        int   sensor_tag;
        int   event_mark;
        ……
}
```

where *sensor_tag* is the sensor ID or geographical location identifier and *event_mark* is the priority level associated with this event. In our design, these two attributes have the following value ranges:

event_mark int 1 – 10
(1: lowest; 10: highest)
sensor_tag int 1 – 32768

These two event attributes are generated by sensor nodes and embedded in the event messages, and eventually carried down to the kernel. Two new system calls, *sys_set_mark()* and *sys_get_mark()*, are added to allow the application server to get/set these two

attributes at the time the service thread is created. We also provide a system call *sys_remove_event* to signal the kernel scheduler to start or stop the flush-out action.

At the application layer, we add two more functional threads, *Watch_ART* and *Flush_Out*, to work together with already existing *Read_Evt* and *Serv_Evt* threads to support the flush-out function. The *Watch_ART* thread keeps watching the system performance by periodically polling and calculating performance metrics, and, if finds the system performance drop below the accepted mark, it signals the *Flush_Out* thread to start the flushing action.

4. Kernel experiments

In this section a series of experiments are performed to assess the effectiveness of the Flush-Out and Early-Drop mechanisms.

4.1 Experiment configuration

The kernel experiments are performed on a cluster system consists of 10 backend nodes and one front-end with an NAT configuration. There are 40 sensor processes running on the boxes separate from the cluster server. The sensor events are formatted byte strings with a fixed length, which carry the information such as sensor ID, IP address, event type, sending time, etc. The two classes of events, i.e. the real-time (RT) events and non-real-time (NRT) events, are sent in 1:1 ratio. The experiments are conducted on Intel Pentium 4 PCs running RedHat Linux O/S with 2.6.10 kernel.

Two configurations are tested: one with a regular kernel and one with a mechanism-enhanced kernel. Other test parameters for the two configurations are the same. Both configurations are tested against the Load Balance (LB) and the Distributed Scheduling (DS) schemes [11] to evaluate the mechanisms' performance under different type of scheduling schemes. The tests are performed against a service time rage of 0.2 – 0.8 seconds.

In the experiment, the events are sent on a data block base, in which each block consists of 100 events. We use following metrics to evaluate the performance of the system:

Event Response Time: defined as the round trip time of an event from the sensor to the cluster server and back to the sensor, and calculated as

$$T_{ave}^{rt} = [\sum_{i=1}^{N} t^{rt}(i)] / N \qquad (1)$$

where $t^{rt}(i)$ is the round trip time of the *i-th* event in the block and *N* is the size of block.

Overall Throughput: defined as the number of events or bits of event data that are serviced by the server per time unit, and calculated by

$$Th = (N * L) / (t_{last\_received} - t_{first\_sent}) \qquad (2)$$

where t_{first_sent} is the time the first event of the block is sent out and $t_{last\_received}$ the last event of the block is received. L is the length (in bits) of the event message.

In addition to average response time and overall throughput, we also use the event drop rate as a performance measurement under high workload. For each sensor event, a timeout is setup and, if the corresponding reply message is not received prior to the timeout, this event is designated as dropped, even though it may eventually come back. The event drop rate is defined as

$$R_{drop} = N_{drop} / N \qquad (3)$$

where N_{drop} is the total number of dropped events in a block and *N* is the block size.

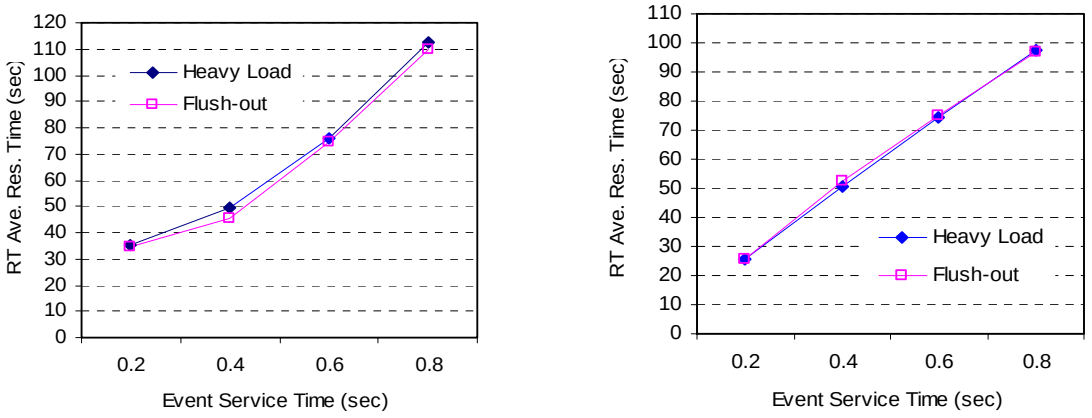
Under such a definition, we shall notice there are two kinds of droppings: dropping at the server due to service unavailability (service dropping), and the events designated as dropped at the receiving end due to timeout (delay dropping).

4.2 Flush-Out experiment

In this test the events are sent with priority levels (RT: 1-20; NRT: 0). When the flush-out action is triggered, the events with priority levels < 6 are filtered out. Figure 3 shows the RT event's average response time against varied event service times, and Figure 4 shows the overall throughput results. We have observed that, under both scheduling schemes, the RT response time and overall throughput numbers are very close for the two configurations.

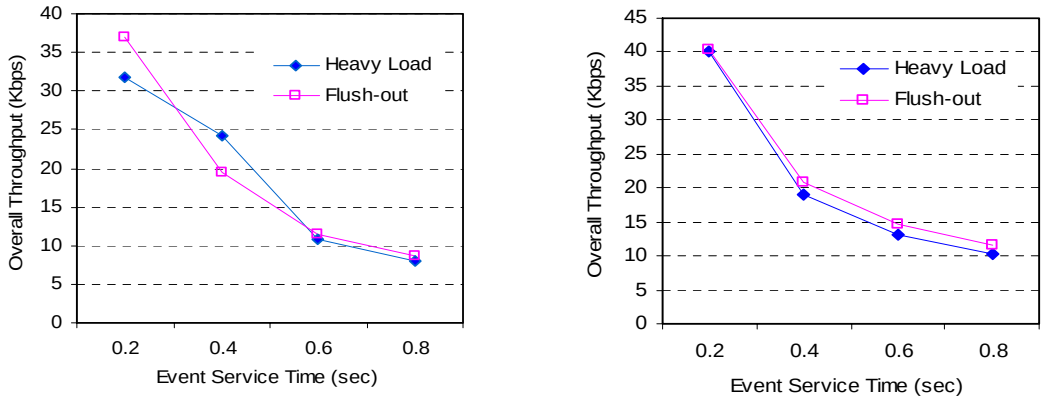
It appears that the flush-out function does not quite help the system responsiveness and throughput, since, under the overloading situations, the available spots in kernel task queues by flushing action are immediately filled up by arriving events. However, the RT event drop rate shows a substantial difference between the kernels with or without flush-out function, shown in Figure 5. Under the LB scheme, the flush-out enhanced kernel subdues the RT drop rate to 3.4%~11.95%, compared to 24.15%~45.40% of regular kernel. Under DS scheme, it is shredded from 24.75%~31% to 7.25%~13.70%, which is an impressive improvement to the system's serviceability. We also observed the system's response to input device (keyboard or mouse) is significantly improved.

With an improved drop rate, much more events get serviced and go through the heavily loaded kernel. The flush-out function achieves this by greatly cutting the delay dropping in a small cost of service dropping. The cause for no big lift in overall throughput is that the increasing number of get-through high priority events is offset by that of dropped low priority events.



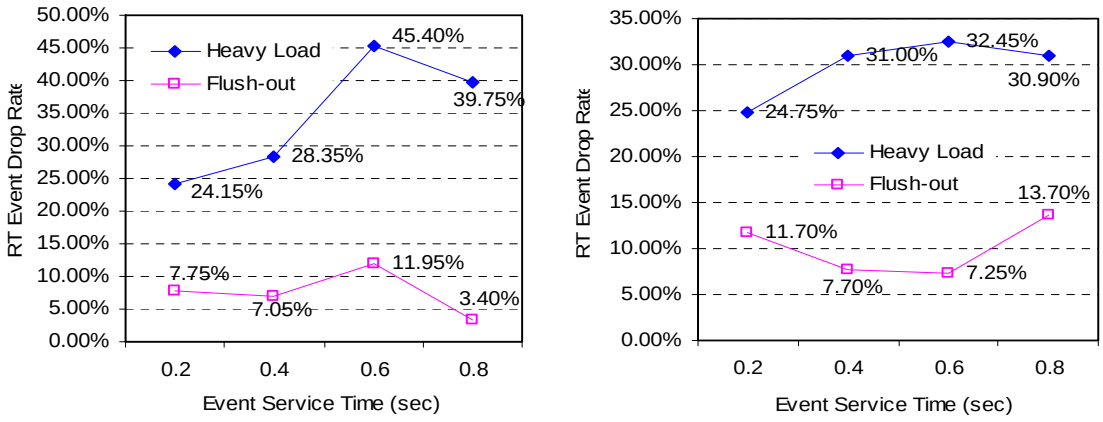
LB Scheme DS Scheme

Figure 3. RT Event Response Time



LB Scheme DS Scheme

Figure 4. Overall Throughput



LB Scheme DS Scheme

Figure 5. RT Event Drop Rate

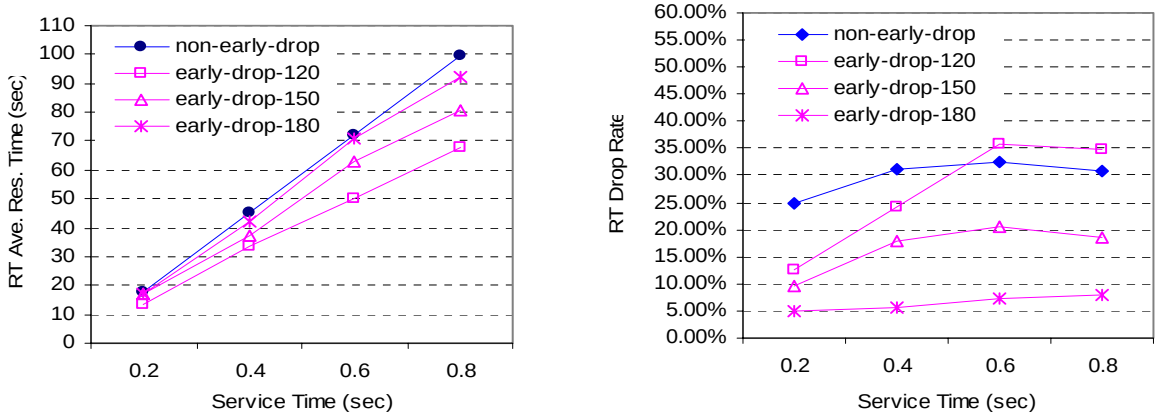
4.3 Early-Drop experiment

The experiment is performed against three configurations. The base configuration is built on top of a regular Linux kernel without early-drop mechanism. The kernel configuration implements the drop action in kernel space (Layer 4 implementation), and the application configuration implements the drop action in application layer (Layer 7 implementation). These three configurations allow us to not only assess the kernel with or without early-drop mechanism, but also to compare the impact of implementing this mechanism in kernel layer or in application layer. In this experiment, the kernel load status is measured by the instant number of active threads in kernel, which can be retrieved by the system call *get_nr_running()*. A threshold RTCS_NR_THRESHOLD is set to determine when to trigger the early-drop operations. In the tests, the threshold values of 120, 150, and 180 are used. A lower threshold value means the system is more sensitive to kernel overloading and tends to drop events earlier while a higher value indicates the system is more tolerable to overloading.

4.3.1 Early-Drop vs. non-Early-Drop

The results of enhanced kernel and regular kernel are shown in Figure 6, in which the early-drop mechanism shows a significant impact to RT event response time and drop rate, but little on the throughput. The number shows, compared to regular kernel, the mechanism enhanced kernel cuts the RT response time by 22%~32% with threshold=120, 13%~19% with threshold=150, and 3%~7% with threshold=180. The RT event drop rate is also impressively dropped to 16~35% for threshold= 120, 9~21% for threshold=150, and 5~7% for threshold=180, compared to regular kernel's 25~33% range. This demonstrates the mechanism can effectively improve system's responsiveness by proactively reacting to overloading situation and avoiding kernel jamming.

It is also noticed that the lower threshold value achieves better response time results in the cost of higher drop rates. For instance, the threshold=120 cuts the response time by 22~32%, but with a drop rate of 12%~35%, compared to Threshold=180's 5%~8% drop rate. Higher threshold values tend to buffer more tasks in kernel, which lead to a lower drop rate and make it more tolerable to high workload. We also compare the results for two groups of events, one with a shorter service time (0.2 sec) and the other with a higher service time (0.6 sec), and list in Table 1. It is observed that both groups yield a decent reduction in response time but the shorter event group gives a much better drop rate. This may indicate the early-drop mechanism works better with short service time events since, under same load level, more of the shorter events get serviced.



RT Response Time

RT Event Drop Rate

Figure 6. Early-Drop vs. non-Early-Drop

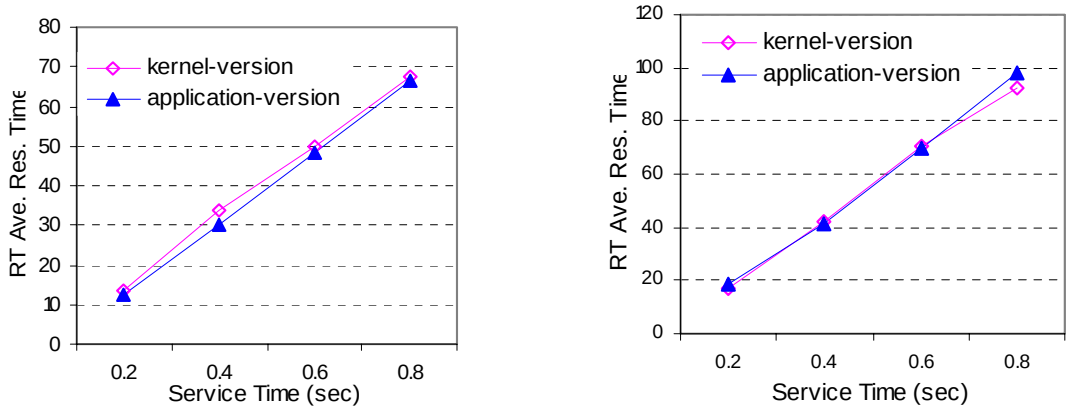
Table 1 Shorter Service Time vs. Longer Service Time (Threshold = 120)

event service time (sec)	RT response time reduction percentage	RT drop rate
0.2	24.73%	12.50%
0.6	30.36%	35.70%

4.3.2 Kernel approach vs. application approach

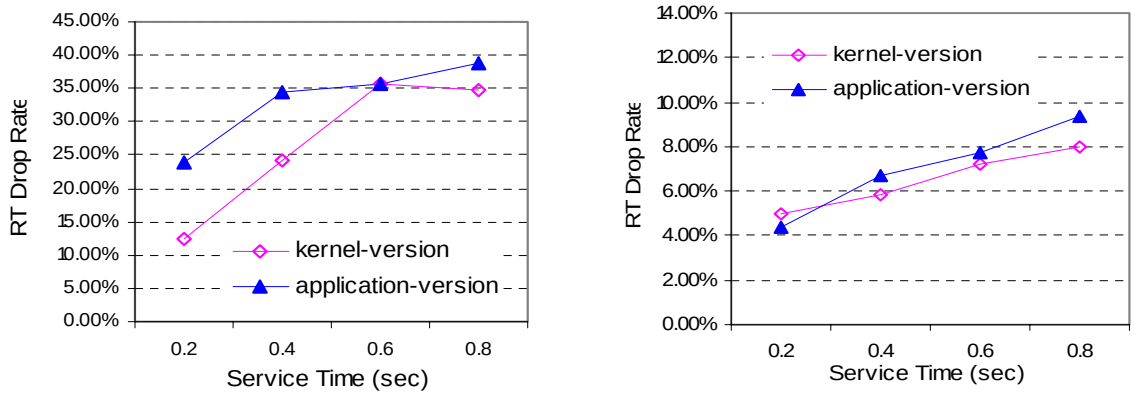
Beside the comparison between the Early-Drop enhanced and regular kernels, we like to probe the difference between the implementations in kernel layer and in application layer. In this test, the two configurations implements same early-drop actions but differ in where the function is placed. We tested the two approaches against two threshold values 120 and 180, with 120 for the system sensitive to overloading and 180 for the system more tolerable to high workload.

The response time results are displayed in Figure 7, which shows the two approaches give very close numbers in response time. The throughput data give the same results. However, the drop rate results in Figure 8 clearly shows the kernel approach outperforms the application approach in all categories except one group (service time = 0.2 sec with threshold = 180). The kernel approach works particularly well with the shorter events (service time = 0.2 or 0.6 sec) in an overloading-sensitive system (Threshold = 120) with a drop rate range 12.50~24.20%, compared to application approach's 23.8~34.4%. The application approach yields slightly better response time result by dropping more events under high load conditions, while the kernel approach gets more events through the heavily loaded kernel by saving the cost in context copying between the kernel and application layers and taking advantage of kernel's fine-grained task scheduling scheme.



Threshold = 120 Threshold = 180

Figure 7. Response Time: Kernel Approach vs. Application Approach



Threshold = 120 Threshold = 180

Figure 8. Drop Rate: Kernel Approach vs. Application Approach

5. Conclusions

The research presented in this paper is targeting on a cluster-based server running under multithreaded mode in which, corresponding to each service request carried by an sensor event, the server directly creates a service thread to serve the event and turns the control of threads to the O/S kernel. The objective of our work is to mitigate the risk of kernel overloading caused by extremely high volume of threads by implementing two application-kernel collaborative mechanisms to proactively respond to load level, which either prevents system from overloading or recovers it if overloading ever happens. Our approach is different from most of the application layer admission control work in embedding the overloading control mechanisms to O/S kernel and focusing on system timely responsiveness under overloading conditions. Embedding these mechanisms in kernel also greatly eases the development of server applications and makes overloading protection more efficient.

The experiment results demonstrate that the Flush-Out mechanism can effectively recover the system from hold-out state and substantially reduce the event drop rate by filtering out certain amount of events from kernel when overloading happens. The mechanism achieves this by getting more events through the jammed kernel, leading to a much lower delay drop rate, however, in the cost of a bit higher service drop rate. The kernel Early-Drop mechanism protects the system from sliding into overloading state by proactively responding to high workload with kernel dropping actions. It also provides the server application a performance tuning knob to choose a shorter response time over a lower drop rate, or vice versa. The comparison between kernel approach and application approach reveals that the kernel approach outperforms the application counterpart in most categories, particularly for the events with short service times.

The application-kernel collaboration functions presented in this paper lift the system's sustainability and serviceability, and improve its timely responsiveness under overloading conditions, which is essential to the applications operating in critical environment. The threshold values used in the Early-Drop mechanism to control the kernel dropping action are trivial, and need further investigation to find its best range. The event service time is an important indicator of the event type, and, in the experiments, we only tested the mechanisms against the service time span of 0.2 - 0.8 seconds. A further test against the range 0.01 - 0.1 second for the very shorter service time events remains to be completed in future.

6. References

[1] J. Nemeroff, L. Garcia, D. Hampel, S. DiPierro, "Communications for Network-Centric Operations: Creating the Information Force". IEEE, Volume 1, 2001 Page(s): 336 – 341.

[2] J.B. Willis, M.J. Davis, "Distributed Sensor Networks on the Future Battlefield", Operations Research Center Technical Report, United States Military Academy West Point, May 2000.

[3] S. N. Simic, S. Sastry, "Distributed Environmental Monitoring Using Random Sensor Networks", *Proceedings of Workshop on Information Processing in Sensor Networks*, Palo Alto, CA, April 2003.

[4] A. Mainwarning, D. Cukker, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless Sensor Networks for Habitat monitoring", *Proceedings of 1st ACM International Workshop on Wireless Sensor Networks and Applications*, pp.88 – 97, ACM Press, 2002.

[5] J. P. Lynch, "Overview of Wireless Sensors for Real-Time Health Monitoring of Civil Structures", *Proc. 4th International Workshop on Structural Control and Monitoring*, New York City, US, June 10 -11, 2004.

[6] J. T. Correll, "From Sensor to Shooter", *Air Force Magazine Online*, Journal of Air Force Association, Vol. 85, No. 2, February 2002.

[7] Sun Microsystems, "Multithreading in the Solaris Environment: A Technical White Paper", http://www.sun.com/software/whitepapers/solaris9/multithread.pdf

[8] V. Shukala, "Linux threading models compared: LinuxThreads and NPTL, IBM develoerWorks", http://www.ibm.com/developerworks/linux/library/l-threading.html

[9] R. Iyer, V. Tewari, K. Kant, "Overload Control Mechanisms for Web Servers", *Workshop on Performance and QoS of Next Generation Networks*, Nagoya, Japan, Nov. 2000.

[10] X. Chen, H. Chen, P. Mohapatra, "An Admission Control Scheme for Predictable Server Response Time for Web Accesses", *Proceedings of 10th World Wide Web Conference*, Hong Kong, May 2001.

[11] Y. Tang, S. Chen, R. Simha, "Scheduling Scheme and Performance Assessment for Timely Responsive Service on Cluster Server", *Proceedings. of IEEE CCGrid 2005*, Cardiff, UK, May 9 - 12, 2005.

[12] T. Voigt, R. Tewari, D. Freimuth, A. Mehra, "Kernel Mechanisms for Service Differentiation in Overloaded Web Servers", *Proceedings of 2001 USENIX Annual Technical Conference*, Boston, June 2001.

[13] R. Tang and R. Simha, "A Delay Differentiation Approach to Real-Time Scheduling on Cluster-Based Multimedia Servers", *Proc. ICT 2002*, Beijing, China, June 2002.

[14] Intel, "Gigabit Technology and Solutions", http://www.intel.com/network/connectivity/resources/doc_library/white_papers/gigabit_ethernet/gigabit_ethernet.pdf