Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 1942–1954
Computer Networks
www.elsevier.com/locate/comnet

Self-prevention of socket buffer overflow

Jin-Hee Choi, Young-Pil Kim, Chuck Yoo *

Department of Computer Science and Engineering, Korea University, A-San Science Building, No. 23, Anam-dong, Sungbuk-Gu, Seoul 136-701, Republic of Korea

Received 18 May 2005; received in revised form 1 October 2006; accepted 3 October 2006
Available online 27 October 2006

Responsible Editor: K. Kant

Abstract

This paper proposes a self-prevention mechanism that architecturally prevents the socket buffer in the networking system from overflowing. By "self-prevention", we mean that the kernel takes certain actions in advance before the kernel gets into an undesirable state, such as thrashing. The shortage of any resource in the kernel may bring the kernel to an undesirable state, and socket buffer overflow is a clear example. First, we explain the reason why socket buffer problem occurs and analyze the impact of each cause through regression analysis. Then, we show how our self-prevention mechanism can minimize the socket buffer problem through simulation, followed by implementation in the Linux kernel.

Keywords: Kernel networking; Socket buffer overflow; Autonomic computing; Proactive resource management

1. Introduction

A good case to demonstrate the need of self-prevention is kernel networking. The states of resources in the kernel networking system change rapidly because packet arrival is generally unpredictable. The socket buffer [5] is one such dynamic resource in the networking system, and socket buffer overflow can occur when the processing speed of the kernel is slower than packet arrival speed. If the socket buffer overflows, applications cannot serve users' requests even though the transport layer protocol guarantees reliable communication. This is obviously an undesirable state of the kernel.

There are two causes of socket buffer overflow. The first is when packets arrive extremely fast. In high speed networks [7], the packet arrival interval is usually very short. Packet arrivals generate interrupts, and during interrupt handling, user processes cannot obtain data from the socket buffer (such behavior is observed regardless of whether the kernel is preemptive or non-preemptive [2]). Second, socket buffer overflow takes place when the packet processing time is too long. Thick protocol stacks increase packet handling time so that user processes have less opportunity to execute the recv( ) system call.

Long packet handling has not been considered a problem until recently; new communication

* Corresponding author. Tel.: +82 2 3290 3198.
E-mail address: hxy@os.korea.ac.kr (C. Yoo).

1389-1286/$ - see front matter
doi:10.1016/j.comnet.2006.10.001

platforms such as software radios [1] are making packet handling time a new problem.

The root of the socket buffer overflow problem lies in the fact that the size of the socket buffer is determined by heuristics when the kernel bootstraps. An intuitive solution is to adjust buffer size by monitoring kernel behavior. However, dynamically adjusting socket buffer is a very expensive operation. In addition, choosing the time in which an adjustment is made is a difficult task. Note that, in this solution, the kernel performs actions after the kernel begins to experience socket buffer overflow.

This paper presents a solution that consists of monitoring kernel behavior and performing lightweight operations in advance. A key operation is to force the kernel to empty the socket buffer while providing unchanged recv( ) semantics. We also provide a method to determine the timing of the operation.

The eventual goal of this paper is to propose a concept, called self-prevention, which can indeed reduce or minimize the chances of the kernel getting into an undesirable state. The socket buffer problem chosen in this paper is an example of an undesirable state. This paper focuses on analyzing and solving the socket buffer overflow problem by forcing the kernel to take action before resulting in socket buffer overflow. We name our solution *packet push*.

The rest of the paper is organized as follows. Section 2 presents an overview of related work. In Section 3, we present basic actions of the traditional UNIX networking system. Section 4 introduces the socket buffer overflow problem and its effect. In Section 5, we elaborate on the analysis of the overflow problem by modeling the UNIX networking system and analyzing the results. Based on the analysis, packet push is presented in Section 6. Section 7 shows the impact of packet push on system performance using simulation study. The implementation architecture of our prototype on Linux and its performance analysis is demonstrated in Section 8. Finally, we conclude this paper in Section 9.

2. Related work

This paper is not the first to architecturally address the socket buffer overflow problem. Druschel et al. [12] observes the overflow problem in high speed networks and proposes an architecture called Lazy Receiver Processing (LRP).

LRP uses two key techniques: lazy protocol processing at the priority level of the receiver and early demultiplexing [13,14].

In LRP, protocol processing for a packet does not occur until the application requests the packet in a recv( ) system call. Packet processing no longer interrupts the running process at the time of packet arrival, unless the receiver has higher scheduling priority than the currently executing process. This avoids inappropriate context switches and increases performance.

The Network Interface Card (NIC) demultiplexes incoming packets by destination socket and places packets directly into a per-socket buffer. This provides feedback to the NIC about a process's ability to keep up with traffic arriving at a socket, when combined with receiver protocol processing at user process priority. Once the socket buffer fills up, the NIC discards further packets destined for the socket until applications process some of the queued packets.

LRP alleviates the socket buffer problem and shares the load, but it cannot completely remove the overflow. This is because the kernel in LRP attempts to solve the problem passively. A better approach to remove the overflow, we think, is that the kernel should attempt to prevent socket buffer overflow more proactively.

Lazy processing has a problem when urgent data is processed too slowly. For example, when real-time constraints [4] are required, packets should usually be handled as soon as possible.

This processing makes time-processing tasks in the protocol difficult. For instance, Round Trip Time (RTT) measurement in TCP [6] would becomes untrust-worthy in LRP.

The proposed solution in this paper is more intuitive and efficient for socket buffer management. Our solution removes socket buffer overflow without any side effects.

3. Traditional network architecture

3.1. Mechanism of traditional receive

On the receiving side, network packet arrivals generate interrupts. The interrupt handler is part of the NIC device driver, and its main job is to encapsulate arrived packets into the shape of network buffers and to put them in an IP queue, which is located between the NIC and IP layer. Then, through an interrupt, the handler notifies the network protocol stack that there is a new packet in the IP queue. The protocol code processes these packets and puts them into the socket's buffers.

After the protocol stack finishes this processing, data copy is performed from the kernel to user address space when the kernel scheduler selects an application with the recv( ) system call, and the packet receiving process eventually completes.

3.2. Mechanism of traditional send

On the sending side, when an application requests the kernel for data transmission by invoking the send( ) system call, the network protocol stack performs transmission differently, depending on the type of the transport layer protocol. That is, if the transport layer protocol is UDP, the payload is wrapped into the network buffer and then passed to the UDP/IP protocol code. When protocol processing is completed, the buffer is again passed to the device driver. If the NIC is in a busy state, the buffer is stored in the NIC interface queue. Then, when the NIC becomes idle, the device driver transmits the network buffer from the queue to the physical network. Conversely, when the transport protocol is TCP, the payload is stored in the socket buffer. This is due to TCP semantics, where the protocol is able to send data only after previous data is acknowledged (ACK). When an ACK is received, the data is sent to the network buffer. Then, TCP manipulates the buffer, and transmits the packets or places them in a queue, in the same way as the UDP.

4. Socket buffer overflow problem

The overflow problem is likely to occur when packet processing is in progress, and packets arrive too frequently. That is, when a new packet arrives before per-packet processing completes, the kernel has to service the interrupt newly generated. As long as new packets are coming in, the kernel cannot execute application code. The socket buffer will finally overflow unless the recv( ) system call moves packets from kernel space. This phenomenon occurs (1) when packets arrive too fast in the system, or (2) when per-packet processing time is too long.

Fast packet arrival has been observed in the area of high speed networks for a long time, but long per-packet processing time is a relatively new problem. Thus, we would like to elaborate on it as follows. Recent advances in communication technology have resulted in the advent of integrated heterogeneous networks. In such networks, new communications platforms are required to be flexible in order to dynamically adapt themselves to changing environments. To emphasize flexibility, many tasks – for instance, modulation, channel coding, MAC processing and so forth – in mobile terminals are implemented in software, such as software radio technology. A problem is that this flexibility comes with additional overhead even though the protocol part such as TCP/IP remains unchanged. That is, while the TCP/IP protocol only deals with the header regardless of payload size, new networking platforms would have a longer delay in per-packet processing because they handle both the payload and header. Without doubt, this increases per-packet processing time.

5. Modeling and analysis of socket buffer overflow problem

5.1. System model

The UNIX networking system can be modeled as a simple queuing system as shown in Fig. 1.

In this system, packets arrive according to a Poisson process [15] of rate λ, so the interarrival times are iid exponential random variables [15] with mean $\frac{1}{\lambda}$. The NIC serves packets with mean rate μ_s and then places the outcomes in N socket buffers. Processes get packets from socket buffers with mean rate μ_p whenever they obtain control of the system. The interarrival and service times are independent, and we assume that the system can accommodate an unlimited number of packets. The size of the

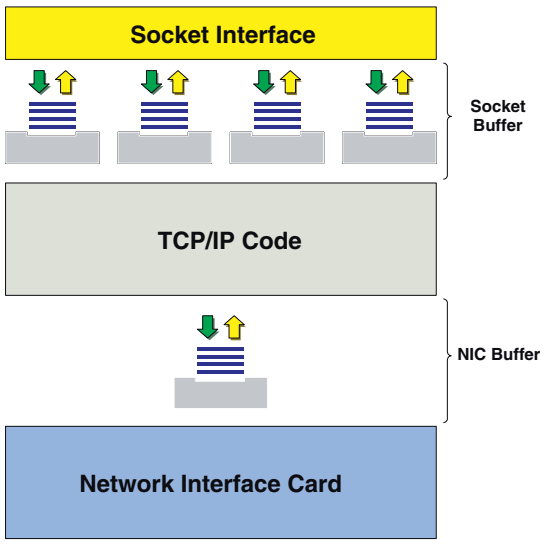


Fig. 1. Network system model.

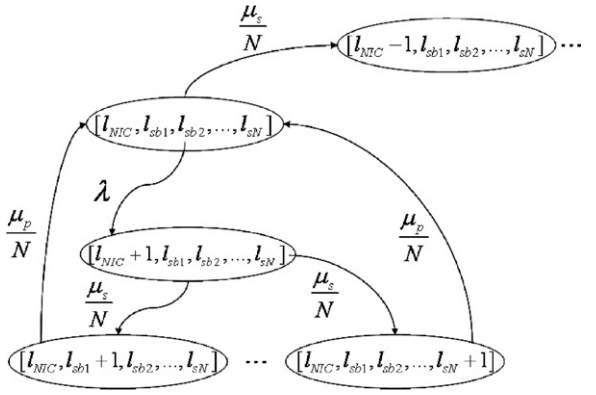


Fig. 2. Markov model.

NIC queue is L_N, and the size of each socket buffer is denoted as L_S. This system can be modeled as a state diagram, depending on the states of the NIC queue and the socket buffer, in which a state would be described as $[S_{\mathrm{NIC}}, S_{sb1}, S_{sb2}, \ldots, S_{sbN}]$. Namely, if a state is modeled as the length of each queue, the state can be expressed as $[l_{\mathrm{NIC}}, l_{sb1}, l_{sb2}, \ldots, l_{sbN}]$[1] and then transition functions that handle events can be defined such as packet arrival, recv( ) execution, and so on. The result is a state machine with a state changing rate that describes the networking system.

This is $(L_N+1)(L_S+1)^N$ states Markov model (see Fig. 2), and it is too complex and inefficient to solve because of the large number of states. Thus, in this paper, system behavior is explained through the simulation result of the above model, using the simulation language, SimPy [8]. The following table summarizes the main configuration factors in our simulation.

The sizes of the NIC queue and socket buffer are based on the values in Linux kernel version 2.4.20 [9]. In the kernel, the size of the socket buffer is set to 64 KB, but, for experimental convenience, the size is set to 64 with the assumption that all packets are of size 1 KB. The number of processes is the same as that of socket buffers, and the default value is set to 3.

In all experiments, the unit of time is ticks, not seconds, and the total simulation time is 300 ticks. The packet arrival interval and system service time are the most important factors in the experiments because the goal of the experiments is to show the impact of the increase in service time on the state of the socket buffer.

[1] Note that l presents the length of current buffer while L is the size of the buffer.

5.2. System behavior

The reason the networking system behaves differently from general queuing networks is that process service rate depends entirely on system state. That is, only when the network system is idle, is the process able to get data from the socket buffer. Even if the system is a non-preemptive kernel, this is still a valid presupposition since system tasks such as packet reception have higher priority than user processes.

Therefore, in the networking system, as the processing delay increases, user processes have fewer chances to access socket buffers, and, as a result, socket buffer overflow occurs even when the system does not have extremely short packet arrivals.

Fig. 3 draws a changing shape of the first socket buffer, which is simulated with initial parameters in Table 1.

In Fig. 3, the X-axis is time (tick) and the Y-axis is the socket buffer size (qlen). There are 2 overflows because max qlen is 64 (Table 1).

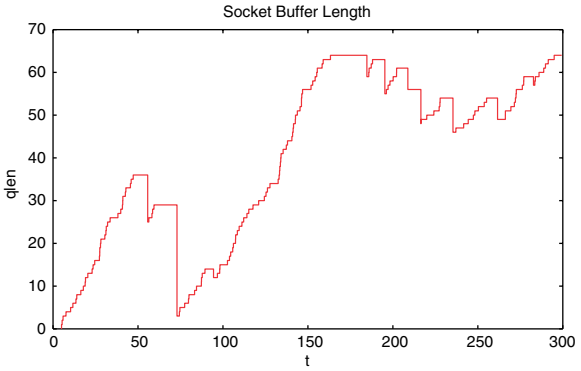


Fig. 3. Buffer length in socket buffer.

Table 1
Configuration factors

Description	Name in model	Name in script	Initial value
NIC queue's size	L_N	NQLen	300
Socket buffer's size	L_S	SQLen	64
Process number	N	procNum	3
Packet arrival interval	$\frac{1}{\lambda}$	arrivalInterval	2.0
System service time	μ_s	serviceTime	5.0

The change in buffer length is a good reflection of the service pattern of processes. A notable tick interval is between 60 and 80, in which qlen dramatically decreases and then gently increases again. This observation says that the service rate tends to fluctuate considerably, and similar saw-toothed inclinations can be seen in several intervals.

A question is whether the number of processes may also have an influence on the socket overflow. However, with the assumption that overhead in context switching is disregarded, only 1 process can be served even when there are N processes in the system and the kernel is in an idle state. Therefore, the number does not have an impact on the performance of the networking system. That is, when it comes to the service rate, the duration of the idle state is important, not the number of processes.

For this reason, the number of processes is excluded as a factor.

5.3. Analysis of variance

With recognition of packet arrival interval and packet transaction delay as the key factors that influence system performance, the degree in which the factors influence the increase in socket buffer length is analyzed in this section using regression model [11].

Table 2 summarizes the experiment results, which have been repeated 1000 times and averaged.

First of all, let us define two variables, x_A and x_B, as follows:

$$x_A = \begin{cases} -1 & \text{if 5 ticks transaction delay,} \\ 1 & \text{if 10 ticks transaction delay,} \end{cases}$$

$$x_B = \begin{cases} -1 & \text{if 2 packet interval,} \\ 1 & \text{if 4 packet interval.} \end{cases}$$

y, the length of the socket buffer, is regressed to x_A and x_B. q_0 is the mean performance of y, and q_A is the effect of x_A. q_{AB} is the interaction between x_A and x_B

$$y = q_0 + q_A x_A + q_B x_B + q_{AB} x_A x_B.$$

Table 2
Experiment results of socket buffer length

Average packet interval (tick)	Packet transaction delay 5 ticks	Packet transaction delay 10 ticks
2	37.86	49.35
4	6.97	23.05

We obtain the following four equations using the above model and Table 2:

$$37.86 = q_0 - q_A - q_B + q_{AB},$$
$$49.35 = q_0 + q_A - q_B - q_{AB},$$
$$6.97 = q_0 - q_A + q_B - q_{AB},$$
$$23.05 = q_0 + q_A + q_B + q_{AB}.$$

By solving the equations, we can get the following regression equation:

$$y \approx 29.31 + 6.89 x_A - 14.3 x_B + 1.15 x_A x_B.$$

From the result, we are able to analyze the respective degrees of impact on buffer length. To begin, total variation (Sum of Squares Total) of y is calculated as follows:

$$\begin{aligned} SST &= \sum_{i=1}^{2^2} (y_i - \bar{y})^2 = 2^2 q_A^2 + 2^2 q_B^2 + 2^2 q_C^2 \\ &= (8.55^2 + 20.04^2 + (-22.34)^2 + (-6.26)^2) \\ &\approx 1012.97 \approx 4 \times 6.89^2 \\ &\quad + 4 \times (-14.3)^2 + 4 \times 1.15^2. \end{aligned}$$

Here, $SST = SSA + SSB + SSAB = 2^2 q_A^2 + 2^2 q_B^2 + 2^2 q_c^2$, and A's impact on total variation is defined as $\frac{SSA}{SST}$.

The analytic degree of the impact can be obtained: the packet arrival interval is 80.75%, the packet transaction delay is 18.75%, and the interaction is 0.52%. This result shows that, as expected, packet arrival interval has a dominant influence on the socket buffer, and transaction time also has a strong impact on the buffer by approximately 20%. The interaction is shown as negligible.

6. Our solution: self-prevention

The approach is to make the kernel manage resources more proactively whereas traditional UNIX behaves passively in response to external inputs like packet arrivals. In the case of the socket buffer, an undesirable state is buffer overflow, and our self-prevention mechanism is to proactively empty the socket buffer in advance before the buffer enters such an undesirable state. For the mechanism, three things must be considered: where to check the state of the socket buffer, when to judge that the buffer would be overflowed in the near future, and how to proactively empty the buffer. We name forced consumption of the whole socket buffer *packet push*. This can be implemented in two different ways.

One way is to give the user process more chance of recv( ) being called (via process scheduler). The solution is based on the fact that, as the recv( ) is called more frequently, socket buffer is well consumed faster so that the buffer length can be reduced. Another way is to remap the socket buffer to user space. The solution is based on the fact that, when the overflow of the socket buffer is predicted, flushing the entire socket buffer prevents buffer overflow.

However, scheduling-based approach has limitations. If the kernel forces the recv( ) calling process to be scheduled before the overflow occurs, the scheduling policy of the process scheduler would impact the behavior of the overall system. So, we felt using scheduler to solve socket buffer overflow would not be practical.

On the contrary, page-remapping [3] uses only virtual memory to handle socket overflow without impacting other parts of the kernel. So this paper uses page remapping to avoid socket buffer overflow.

6.1. Check point: where to check the overflow

To check whether the buffer would be overflowed, the check point in the kernel must first be determined.

A check point is placed between the link layer and network layer of the protocol stack, as shown in Fig. 4. The intention of the placement is to avoid unnecessary check point overhead. This is because only the packets that pass through-the check point can make impact on the growth of the socket buffer.

Then, whenever a packet passes through the check point, the kernel checks whether there is a socket buffer that has length over *ov_thresh* (overflow threshold to be discussed in the following section). If there is, the kernel directly pushes data to avoid socket buffer overflow without waiting for consumption by user process.

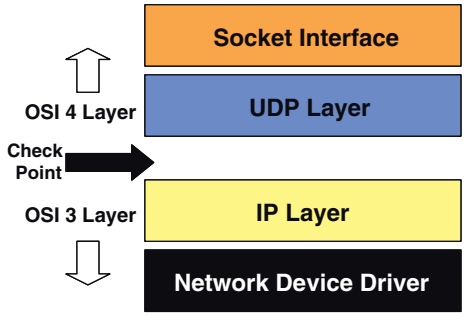


Fig. 4. Check point in the kernel.

6.2. Threshold: how to determining the overflow

To obtain the proper *ov_thresh*, two values are tracked: the mean of packet arrival interval, $\frac{1}{\lambda}$ and mean time of the packet push operation, m. A good threshold adapts itself to changing m and λ values, and eventually prevents packet push from being performed needlessly.

The threshold starts with $\frac{2L_S}{3}$. So, when the buffer length is greater than $\frac{2L_S}{3}$, packet push is performed, and the kernel calculates the moving average of the packet push operation time, m. When the new packet push operation time is obtained, the kernel recalculates *ov_thresh* as follows:

$$ov\_thresh = \mathrm{MIN}\left(\frac{2L_S}{3}, L_S - \lambda m\right).$$

Note that λm is the estimated amount of the incoming packet during the packet push operation. Pseudo code of this mechanism is presented in Fig. 5.

A key of Fig. 5 is a slow filter that is used to obtain the moving average of the packet push time (old value is more weighted than new value).

However, in getting λ, a fast filter is used. The reason is that the packet arrival rate is a time critical value while the packet push time is generally stable.

6.3. Packet push: how to empty the buffer

Fig. 6 shows the conceptual shape of the system during the packet push process. As previously mentioned, the core role of the packet push is that the kernel reduces the socket buffer overflow from too frequent packet arrivals or too long packet transaction delays.

This paper uses page remapping to implement packet push. This section explains packet push

```
1. α=0.2;
2. ov_thresh=2L_S/3;
3. whenever(L_S^cur > ov_thresh){
4.     packet_push();
5.     m = α × new_m + (1 − α) × m;
6.     ov_thresh = MIN(2L_S/3, L_S − λm);
7. }
```

Fig. 5. Pseudo code of overflow threshold.

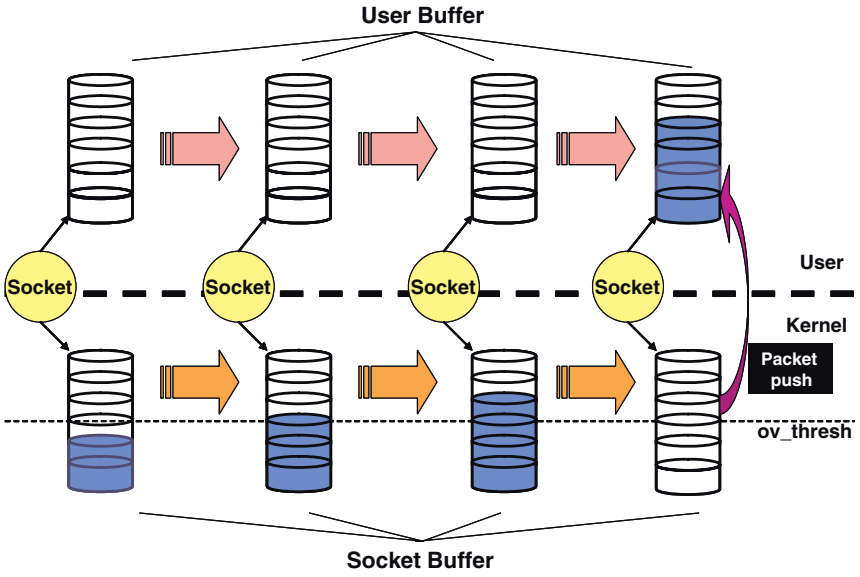


Fig. 6. Push process.

implementation in Linux with details of socket buffer structure.

Conceptually, the queue (or buffer) consists of continuous memory blocks in a memory area, and packets are copied into the area. In Linux, however, the socket buffer is implemented by chaining several skbuff objects. Whenever the packet arrives, the kernel obtains a skbuff object from the slab cache [16] and adds the object to the tail of the socket buffer. The max size of the socket buffer is controlled by a limit value, 64 KB. Fig. 7 shows the socket buffer structure where pointers such as 'next', 'prev', 'head' and device information such as 'dev' and 'data' of the socket buffer are scattered, unlike the mbuf structure of BSD. Page remapping is performed only for the data area (refer Fig. 8) because applications only need the data.

After such remapping completes, the kernel marks the *pkt_push_done* field of task_struct in Linux to indicate that the packet push completes. Before copying data, recv( ) checks the *pkt_push_done* field, and when the field is marked, recv( ) copies the data from the designated mapping area('user buffer start address' in Fig. 8) to the heap address of recv( ) parameter (user-to-user copy). Here, the designated mapping area means that a user memory area where the received data is temporarily mapped. To make user-to-user copy, the virtual address for the remapped data is required, and for this the *uva_data* field is added to the skbuff structure. Using this field, recv( ) can simply use memcpy( ) to copy the remapped data to the user-requested heap area. If the *pkt_push_done* field is unmarked, recv( ) performs as the original recv( ).

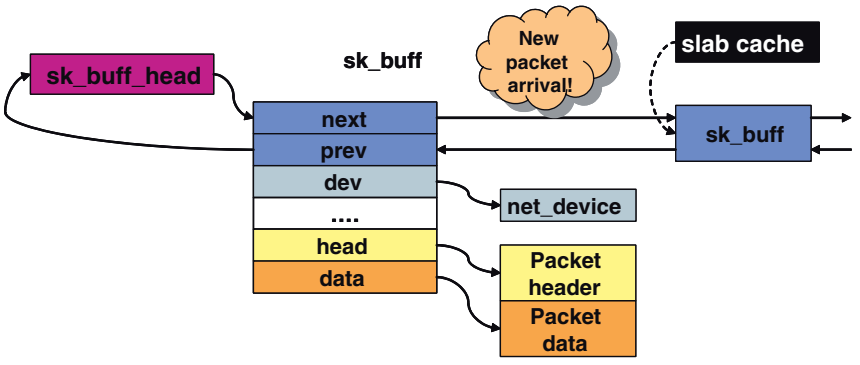


Fig. 7. Structure of socket buffer in Linux.

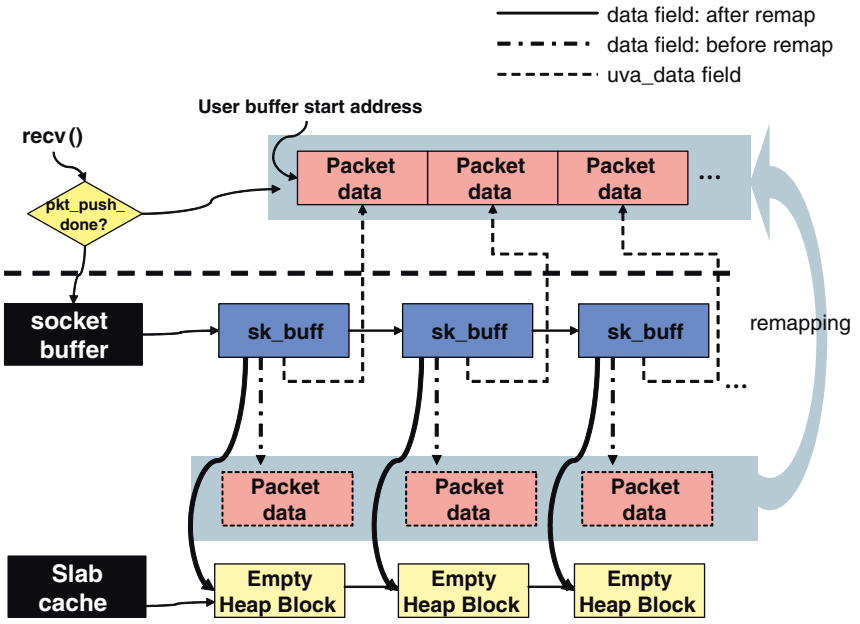


Fig. 8. Data remapping process.

7. Simulation result

The goal of this section is to show the impact of the packet push operation on the system through simulation.

The frequency of overflow depends on the two factors: packet transaction delay and packet arrival rate.

Fig. 9 shows the relation between the two key factors. The *X*-axis of the graph represents the packet transaction delay, and *Y*-axis depicts the frequency of overflow in the socket buffer. The result in Fig. 9 shows that, in general, frequency of overflow is proportional to packet transaction delay. However, in the case of 1 tick as packet arrival interval, the degree of increase of time is small, which means the impact of the arrival interval overwhelms that of the transaction delay, and consequently the influence of the transaction delay does not attract attention. In addition, when the arrival interval becomes too long, overflow does not occur at all since there are not enough sufficient data for overflow to occur (see case "interval(4)" in Fig. 9). Overall, excluding above two cases, the number of overflows is in proportion to packet transaction delay in Fig. 9. In contrast, in the packet push system (UNIX with packet push), overflows did not occur in any case because the kernel handles buffer overflow in advance.

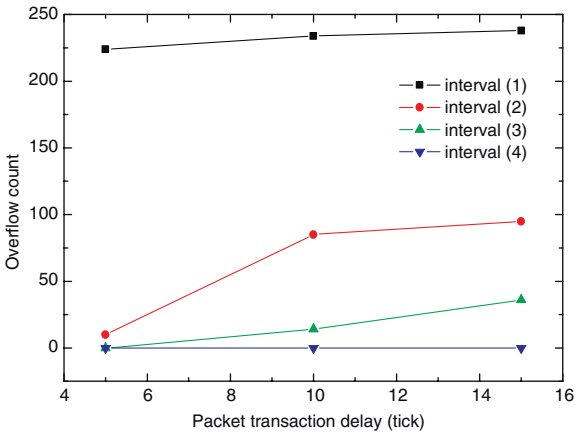


Fig. 9. Relationship between the arrival interval and the transaction delay.

Fig. 10 (with packet push) illustrates the changing shape of the socket buffer length under the same conditions (the arrival interval is 2 ticks, and the packet transaction delay is 5 ticks) with Fig. 3 (without packet push). The difference between the two figures is well shown after 150 ticks. In the UNIX system, because the kernel does not perform any specific operation even though overflows occur, overflows continuously occur, and also the socket buffer has a long mean length. (Fig. 3). However, the packet push system shows stable behavior, and

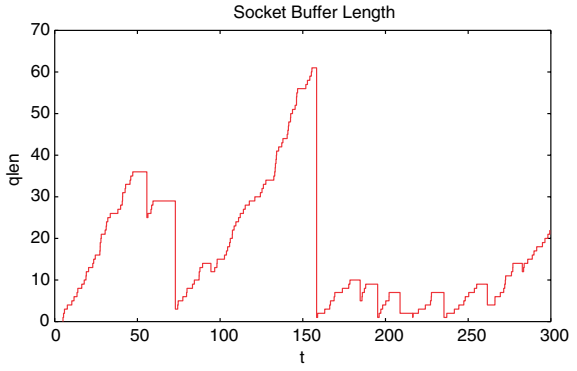


Fig. 10. Length variation in socket buffer under the packet push system.

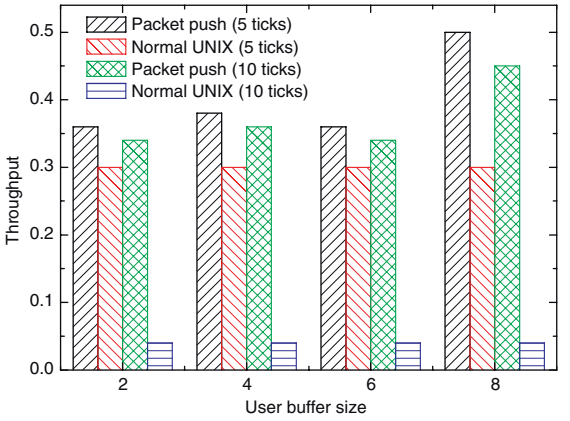


Fig. 12. Throughput comparison between the packet push and UNIX systems.

its socket buffer has a relatively short mean length since the kernel flushes the buffer in advance whenever the buffer length goes over the threshold.

Fig. 11 shows the system behavior well when the packet transaction delay is long (10 ticks). Fig. 11(a) shows the buffer length variation in UNIX. In this case, overflow occurs in the socket buffer just before 150 ticks. After 150 ticks, the UNIX system cannot handle the increasing load adequately, and as a result the kernel cannot avoid socket buffer overflow. Conversely, in Fig. 11(b) with packet push system, we can see that the kernel pushes the packets periodically and maintains the saw-tooth shape of the socket buffer. Consequently, the packet push system does not overflow the socket buffer, and the buffer length can always be kept relatively short.

Another notable point in Fig. 11 is that the kernel shifts all packets to the user buffer whenever the buffer length rises above the threshold. However, since the parameter of the recv( ) system call determines the size of the data being read, only the data of MIN (the length of the socket buffer, the length of the user buffer) can be pushed in the real system. It is natural that, the throughput of the user process is proportional to the size of the pushed data.

Fig. 12 shows the throughput of the user process with varying user buffer sizes. The *X*-axis of the graph is user buffer size, and *Y*-axis is the number of the user process' taking-events during the simulation time. All experiments are performed with packet arrival intervals of 2 ticks, and transaction delays are 5 and 10 ticks.

The experiments show that the packet push system shows better performance in terms of throughput, than the UNIX system.

In particular, the UNIX system shows a dramatic performance drop of 88.2%, when increasing the packet transaction delay from 5 to 10 ticks, while the packet push system shows a much smaller reduction of 16.7%, under the same increase in packet transaction delay.

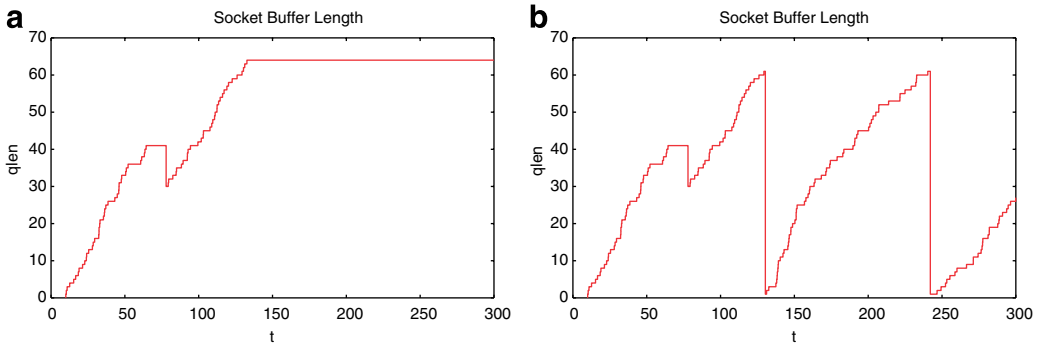


Fig. 11. Comparison of buffer length variation when the packet transaction delay is 10 ticks. (a) UNIX system and (b) packet push system.

8. Experimental result

8.1. Experimental environment

The packet push prototype is implemented as a Linux Kernel Module (LKM) device driverr [10] in Linux kernel version 2.4.20-8 [9]. The push prototype consists of the push device driver and push application that is used in testing. The key role of the device driver pushes the kernel buffer to the user address by remapping the pages. In addition, it issues the packet arrival interrupt and checks whether the buffer length goes over *ov_thresh*. The goal of this prototype is to show that packet push could be easily applied to a real system, and finally could prevent the buffer from overflowing effectively.

The following Fig. 13 is the implementation architecture for the push application and device driver, and we present the source code in "http://os.korea.ac.kr/network/drone/pktpush.tar.gz".

The interaction between the application and push device driver is performed by overriding the ioctl( ) operation. The push application opens the push device, and the application does the following ioctl( ) operation using its descriptor.

#define PUSH_KMALLOC	_IO (PUSH_IOCTL_MAGIC, 1)
#define PUSH_KFREE	_IO (PUSH_IOCTL_MAGIC, 2)
#define PUSH_TIMER_INIT	_IO (PUSH_IOCTL_MAGIC, 3)
#define PUSH_RESET	_IO (PUSH_IOCTL_MAGIC, 4)
#define PUSH_SET_TIMER_ITERATION	_IO (PUSH_IOCTL_MAGIC, 5)
#define PUSH_SET_THRESHOLD	_IO (PUSH_IOCTL_MAGIC, 6)
#define PUSH_RECV	_IO (PUSH_IOCTL_MAGIC, 7)
#define PUSH_MAPPING	_IO (PUSH_IOCTL_MAGIC, 8)

The push application configures several parameters for experiments when opening the push device.

The parameters are PUSH KMALLOC (the size of kernel buffer), PUSH SET TIMER INIT (the packet arrival rate), PUSH SET TIMER ITERATION (ongoing packet count), and PUSH SET THRESHOLD (the overflow threshold). On setting the initial parameters, the application requests the PUSH RECV operation to obtain the packets, and waits. After this, when the push device driver performs a push operation and returns a pointer to the user buffer, the application escapes from waiting and display the virtual address and content of the new buffer. This address points the kernel buffer that is remapped to the user space. Note that the copy operation is not performed during the push phase. The remapped user buffer is not influenced even when the PUSH KFREE operation frees the kernel buffer.

After receiving the ioctl( ) operation from the application, the push device driver handles the requests. To emulate the packet arrival event, we put the timer in the driver. The timer interval is a Poisson process of rate PUSH SET TIMER INIT, and for experimental convenience we limit the number of the packet arrival events, PUSH SET TIMER ITERATION.

The push_timeout( ) function increases the push_counter whenever the packet arrival event is generated. When push_counter reaches the overflow threshold, the push device driver performs page remapping to a designated mapping area of user address space. For implementing memory mapper (do_push_mmap( )), we modified the sys_mmap( ) function to support the file structure of the push device driver. The do_push_mmap( ) function registers the page handler(push_vma_nopage( )) which updates an old page table. After push operation, the push counter is initialized to 0, and whenever timer events occur, the counter increases repeatedly.

8.2. Overhead comparison

We perform an experiment to measure the time of recv( ) in both the packet push and the original Linux mechanism. For overhead measurement, we used a well-known network performance benchmark tool, called 'netperf'. The client of 'netperf' runs Linux 2.4.20, and the server runs a modified Linux that implements packet push. It is assumed that the test data has been measured in a real

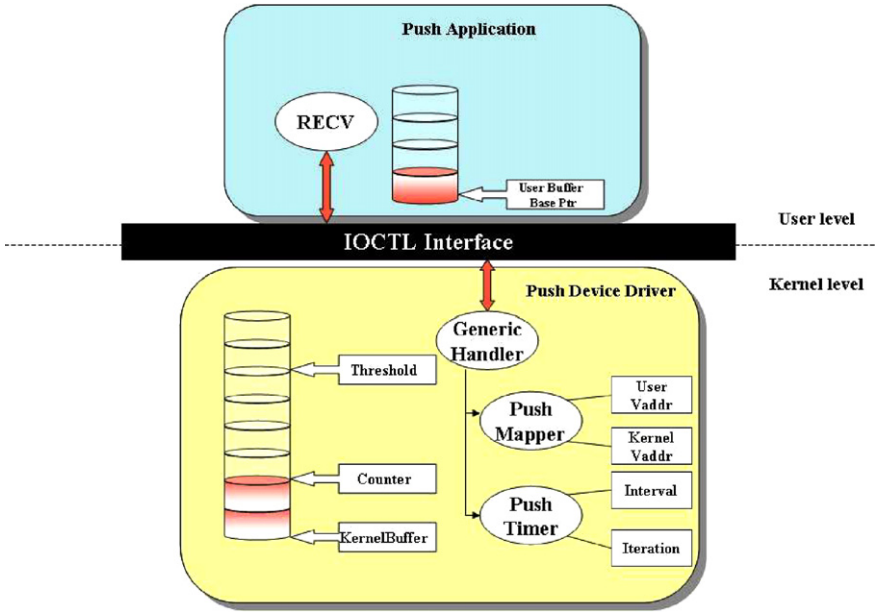


Fig. 13. Implementation architecture for the push system on Linux.

Table 3
The specific breakdown of the receive overhead

Overhead	Description
a. User level latency	The elapsed time between invoking of recv( ) system call and returning of it
b. Socket buffer consumption time	The time needed to consume a 'skbuff' list by an application process when the scheduler gives a chance of resuming recv( ) system call
c. Socket buffer node fill-up time	The time needed to fill a 'skbuff' node that is in-kernel data structure with packet data that is transferred from NIC
d. Physical network transfer time	The time that receiver NIC gets packets through physical network

environment. Regarding cache aspect, when a packet arrives, the DMA controller passes the packet from the NIC to kernel buffer without going through the processor cache. The measurement data is not affected by the processor cache.

In current Linux kernel implementation (kernel version 2.4.20), we can break down the overhead of recv( ) in the following steps (see Table 3). The receive overhead consists of four steps in Table 3. With heavy network traffic, the kernel is busy with frequent interrupt handling so that the kernel cannot give enough chances to consume packet data in the application. The push operation proposed by this paper improves '*b*' and '*c*' in the table. In push operation, the buffer overflow is estimated, and using the estimation results, we prevent unnecessary packet drops so that '*b*' improves. In addition, memory mapping saves data copy, which reduces '*c*'.

It turns out, as in Fig. 14, that packet push using the page remapping dramatically reduces the recv( ) overhead to take the data from the socket buffer (over 80).

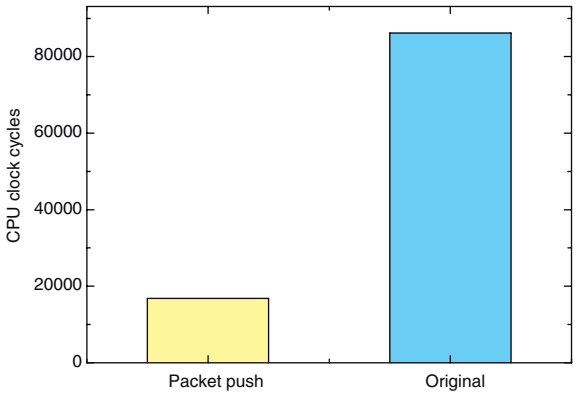


Fig. 14. Comparison of recv( ) overhead.

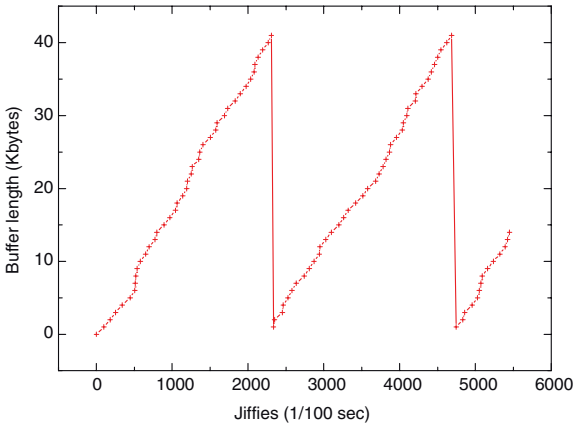


Fig. 15. Socket buffer length of prototype.

8.3. Prototype behavior

Fig. 15 shows the new shape of the socket buffer length in our implementation. This is quite similar to our simulation result in Fig. 11(b). In Fig. 15, the unit of *X*-axis is jiffies. The jiffies global variable stores the number of elapsed ticks since the system started. It is set to 0 during kernel initialization and incremented by 1 when a timer-interrupt occurs. Because the default value of the timer-interrupt interval in Linux is 10 ms, jiffies are updated every tick.

From the prototype's behavior, we can say with fair certainty that the packet push mechanism stabilize the kernel from socket overflow.

8.4. System throughput

We also conducted an experiment to compare throughput of our prototype with that of Linux. For this experiment, a pseudo device that artificially generates packets is implemented. The packet push is performed whenever the kernel buffer rises above *ov_thresh* results from the generated packets. Throughput is measured by the amount of data to move from the buffer to heap area of user space during limited time. Fig. 16 shows two throughputs, and the result of the simulation and the experiment shows nearly the same tendency. Also note that packet push affects only the processes that have packets in the kernel. Therefore, it minimizes the impact on the overall system. By removing the socket buffer overflow, the overall system is stabilized even with heavy network traffics.

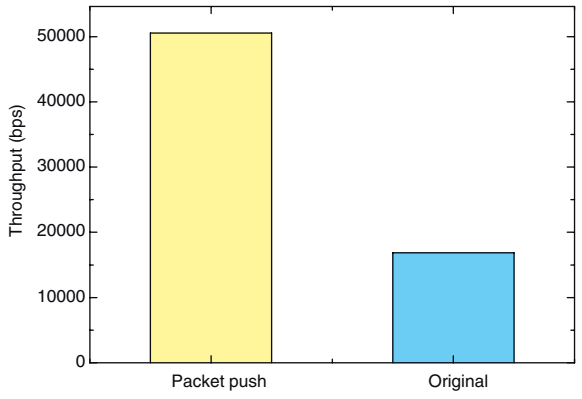


Fig. 16. Throughput comparison of prototype and Linux.

9. Conclusion

Self-prevention is to take certain actions 'in advance' before the kernel gets into an undesirable state. In this paper, the socket buffer overflow problem is addressed and the packet push technique is proposed as a solution. The packet push is a form of self-prevention for the socket buffer overflow problem. The packet push prevents the socket buffer overflow by flushing the buffer in advance, and the results of the simulations and implementation are presented. The packet push is implemented on Linux, and page-remapping is used to flush the socket buffer.

In addition, the result of the measurement of our prototype shows that packet flush with page remapping consumes only 20% of CPU cycles of the original Linux mechanism.

Acknowledgement

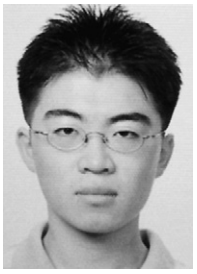
This work was supported by Grant No. R01-2004-000-10588-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

References

[1] S. Srikanteswara, R. Palat, J. Reed, P. Athanas, An overview of configurable computing machines for software radio handsets, IEEE Communications Magazine 41 (7) (2003).
[2] D. Bovet, M. Cesati, Understanding the Linux Kernel, second ed., O'Reilly, 2002.
[3] Hsiao-Keng, J. Chu, Zero-copy TCP in Solaris, in: USENIX 1996 Annual Technical Conference.
[4] J. Liu, Real-Time Systems, Prentice-Hall, 2000.
[5] M. McKusick, G. Neville-Neil, The Design and Implementation of the FreeBSD Operating System, Addison-Wesley, 2004.

[6] D. Comer, Internetworking with TCP/IP, Principles, Protocols and Architecture, vol.1, fourth ed., Prentice-Hall, 2000.
[7] S. Feit, Local Area High Speed Networks, Pearson Education, 2000.
[8] K. Muller, T. Vignaux: SimPy: Simulating Systems in Python, ONLamp.com Python Devcenter, 2003.
[9] Linux kernel 2.4.2, http://lxr.linux.no/source/?v=2.4.20.
[10] J. Corbet, A. Rubini, G. Kroah-hartman, Linux Device Driver, third ed., O'Reilly, 2005.
[11] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons, 1991.
[12] P. Druschel, G. Banga, Lazy receiver processing (LRP): a network subsystem architecture for server systems, USENIX OSDI 1996.
[13] C. Maeda, B.N. Bershad, Protocol service decomposition for high-performance networking, in: ACM SOSP 1993.
[14] C. Thekkath, T. Nguyen, E. Moy, E. Lazowska, Implementing network protocols at user level, in: SIGCOMM 1993.
[15] A. Leon-Garcia, Probability and Random Processes for Electrical Engineering, second ed., Addison-Wesley, 1994.
[16] J. Bonwick, The slab allocator, an object-caching kernel memory allocator, in: USENIX 1994 Annual Technical Conference.

Jin-Hee Choi received his B.S., M.S., and Ph.D. degrees in computer science from Korea University, Seoul, Korea, in 2000, 2002, and 2006 respectively. Currently he is with Telecommunication R&D Center, Samsung Electronics. His current interests are in kernel networking and mobile software platform.



Young-Pil Kim received his B.S. and M.S. degrees in computer science from Korea University, Seoul, Korea, in 2002 and 2004, respectively. He is currently a Ph.D. candidate at Korea University, Seoul, Korea. His current interests are in the kernel reconfigurability and kernel networking.

Chuck Yoo received his B.S. degree in electronic engineering from Seoul National University, Seoul, Korea and the M.S. and Ph.D. in computer science in University of Michigan. He worked as a researcher in Sun Microsystems Laboratory, from 1990 to 1995. He is now a Professor in Department of Computer Science and Engineering, Korea University, Seoul, Korea. His research interests include high performance networks, multimedia streaming, and operating systems. He served as a member of the organizing committee for NOSSDAV 2001.