Analysis of Linux UDP Sockets Concurrent Performance

Diego Rivera*, Eduardo Achá†, Javier Bustos-Jiménez‡ and José Piquer†
*Institut Telecom SudParis, France. Email: diego.rivera-villagra@telecom-sudparis.eu
†DCC, Universidad de Chile. Emails: {eacha,jpiquer}@dcc.uchile.cl
‡NIC Chile Research Labs, Chile. Email: jbustos@niclabs.cl

***Abstract*—Almost all of DNS queries that traverse Internet are transported via UDP in self-contained small packages. Therefore, with no restriction of packet ordering, the intuition would say that adding thread-parallelism to the servers will increase their performance, but it does not.**

This paper study the problem of serialization accesses to UDP sockets, and states the problem in the way the packets are enqueued in the socket at kernel level, which introduces high levels of contention in synchronization primitives for thread using. As a naïve solution, we present a multi-queue receiver network stack which improves the performance of processing UDP small packages when multiple threads read from the same socket.

I. Introduction

With the growing of *Internet* the efforts in research and development has been focused in manage and communicate high amount of (big) data. Some of these efforts included the use of parallel computing (threads) in server-side softwares, aiming to adapt the server to the steadily growing demand.

Nevertheless, Facebook [10] and Toshiba [8] have showed that introducing threads to server-side applications, such as *memcached* and *BIND* respectively, do not necessarily imply a gain in the number of queries the server is capable to handle per time unit. Their engineers stated the problem inside the Linux kernel (which was the Operating System used in both studies), but leaving the analysis to later works, due its complexity.

The same behavior has been analyzed by the engineers of *MOSBENCH* Project, by performing benchmark tests with well-known non-scalable applications (such as *memcached* and *Apache*) [4]. The results of these tests showed that many optimizations can be done in the application itself, but some other servers might be bottlenecked by the I/O operations. A further analysis testing scalable and non-scalable lock types in Linux kernel, led the team to conclude that "traditional kernel designs might be compatible with achieving scalability on multicore computers". Moreover, these results explained mathematically and experimentally that the use of non-scalable locks may yield in whole system performance collapse.

Above studies lead to suspect in the Operating System as the root of the problem, which is not able to provide a scalable interface for network communications. Thus, a completely scalable network stack would solve the performance issues already observed.

Our main contribution is the study of this non-scalable-threads issue in recent versions of Linux kernel is state the problem in the way packets are enqueued in the socket, which introduces high levels of contention in synchronization primitives due threads. We also provide a naïve solution based in multiples queues per socket, intended for servers whose queries are small and self-contained in unique packets, such as DNS Servers. We show that this solution improves the performance of parallel-thread reads from a socket.

This article is organized as follows: Section III shows a small introduction to how Operating Systems receive a packet from the network. Section IV reproduces the problem, and states it inside the kernel. Section V studies the spinlocks used by the socket to enqueue the incoming packet right after these have been processed through the network stack, and states the problem in the synchronization schema used by data structures. Finally, an implementation of a solution for DNS services is presented, followed by our conclusions and future work.

II. Related Work

The impact of locking in multithreaded network protocols has been studied since around 20 years. For instance, Björkman and Gunningberg [2] studied the effects of locking systems in the implementation of UDP and TCP over IP and ETH protocols in the x-kernel emulator, previously presented by Hutchinson and Peterson [7]. Björkman and Gunningberg also implemented a parallel version of this emulator, measuring the performance gain achieved by using multiple processors when processing incoming and outgoing packets. This measurements also lead them to identify some bottlenecks inside x-kernel such as, for instance, copying information from device to kernel memory.

The above work was extended by Nahum et. al. [9], running a parallel version of x-kernel in Sillicon Graphics multiprocessors and analyzing the effects of checksumming, ordering and locking in parallel TCP implementation. They conclude that a simpler locking system and the use of atomic primitives can make a big difference in performance, which pursues to avoid at maximum the contention created by locking.

These studies were continued with measurements made by Schmidt and Suda [11] in SunOS using ASX Framework. They emulated two types of message parallelism architectures: connection and message-based. They conclude that a message-based parallelism is more suitable for connectionless applica-

tions, such as DNS Servers. Moreover, they state that synchronization costs have a substantial impact in performance, thus selecting the incorrect synchronization primitive would decrease significantly the throughput obtained.

More recently, Willman et. al. [12] studied the effects of connection-based parallelism implemented by default in FreeBSD Operating System, measuring the throughput and scalability in the system and how these metrics are influenced by locking, cache behavior and scheduler overheads. These measurements allowed them to conclude that a uniprocessor version of FreeBSD kernel degrades its performance as long as additional connections are added.

In the same line, Han et. al. proposed *MegaPipe, an API for Scalable Network I/O* [6]. This new API allowed some server softwares (like *memcached* and *nginx*) to reach up to 75% of gain in throughput. Moreover, they found that message-oriented connections with small messages incur in greater overheads that connections with larger messages. A typical example of these kind of services is DNS, which runs over UDP with small-sized queries.

In our work, we aim to extend the research previously done by studying the Linux Kernel, in order to isolate the root of the serialization accesses to UDP sockets previously observed by Toshiba when trying to scale BIND server. This is commonly identified as the sign of non-scalable I/O operations when adding threads to listen in a single open UDP socket.

III. Linux network stack working

Sending or receiving a packet through network devices involves a big amount of processing before being sent or after arrived to the host. In this chapter we will explain the mechanism used by Linux when a packet is sent to the network or received from it.

Sockets in Linux are a kernel data structure consisting in: a state field, a type field, a flags field, a pointer to an instance of `struct proto_ops` (containing pointers to functions which implement sockets' operations) and a pointer to a `struct sock` instance. This last structure is relevant to this study, because it defines fields related with reception and transmission of packets:

- *A structure* `socket_lock` containing a spinlock used to lock the entire socket.
- *Three packet queues (reception, transmission and backlog)* used to store packets before being sent (in case of transmit information to other host) or after received (in case of incoming packets from the network). Backlog queue is used in special cases where the receiving queue is locked, and its working will be explained in section V-A. Each one of these lists is protected by a spinlock which guarantees its consistency against concurrent accesses.
- *Memory Accounting fields* used to keep track of the amount of memory being used by the socket and, specially, the amount of data in the queues. This feature is used to avoid the socket fill up the entire memory, leading to Operating System malfunction.

A. Packet Receiving

Figure 1 represents the main tasks performed by Linux when a packet arrives to the Network Interface Card (NIC). The process is performed in the figure from bottom to top.

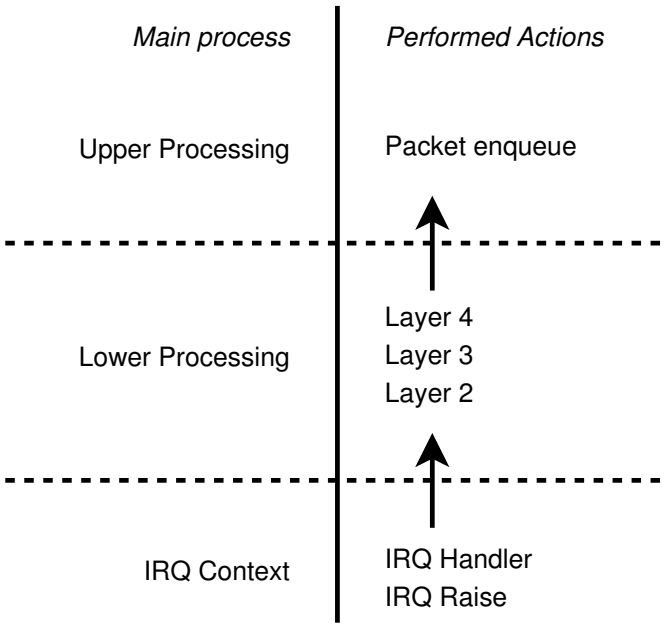

Fig. 1: Packet receiving process

Three main tasks can be identified in the process: (1) *IRQ Context*, started when the packet arrives to the NIC, raising a Hardware Interruption (HardIRQ) and ended with the schedule of a Software Interruption (SoftIRQ); (2) *Lower Processing*, consisting in the execution of the SoftIRQ previously scheduled, which process the packet through Layers 2 (L2) to 4 (L4); and (3) *Upper Processing*, where the packet is enqueued in one of the socket's queue prior to being read by the application. This design is intended to minimize the time spent by the processors handling HardIRQs, deferring the biggest part of the work for a future instant[1].

In the following sections we will study how the synchronization primitives can impact in the general performance of receiving a packet. Further information and details about how the packet is processed in each step when transmitting and receiving packets can be found in [1], [13].

B. Memory Accounting

Each time some information is sent or received by the kernel, it has to be copied into kernel's memory in order to be preprocessed before being sent through the device or read by the user. Linux limits the amount of memory a single socket can use to store packets inside the kernel, avoiding scenarios where packets overflow kernel's memory.

This feature is known as *Memory Accounting* for sockets. In the special case of UDP, it was cloned directly from the TCP implementation, including the locking system used to keep the data consistent. This extra synchronization introduced a new spinlock in the sockets, which raised UDP latencies especially

when using this protocol with multicast. This behavior was observed by the community, who developed patches to the kernel in order to deactivate Memory Accounting for UDP sockets [5].

IV. UDP PERFORMANCE IN LINUX

As a first step, we measure the actual performance of UDP sockets against concurrent read accesses. To achieve this goal, we must isolate the portion of code we want to test.

Linux provides the loopback virtual interface as a simple way to communicate two applications in the same machine, using the network stack and with the advantage of avoiding any overhead introduced by HardIRQs. This last reason combined with the goal of analyze only network's stack performance, lead us to select loopback interface to measure the time spent and performance of network stack's code. In addition, and as a reference point, we compare the UDP socket performance against a kernel FIFO queue provided by the `mkfifo` tool.

In order to stress the structures and determine its behavior in parallel scenarios, concurrent reads will be performed with no synchronization between application threads, thus time measurements will represent the behavior of each source in a scenario where the amount of threads exceeds the amount of processors available of the system.

More in detail, we measured the amount of time required to read 500.000 information units of 10 bytes each one, simulating a constant incoming of small messages from a client to a server; a common scenario for DNS servers. The volume of reading threads in server-side will be doubled each time, until the amount of available processors is exceeded, simulating scenarios varying from a very small amount of reading threads, up to others where reading threads overwhelm processors. Both applications will run in the same machine, using different mechanisms to transfer the information from client to server: a UDP socket connected through loopback interface and a FIFO queue created in the file system. For these tests, we used a Dell Optiplex 990, Intel Core i5-2400 processor (4 logic cores) and 8 GB DDR3 RAM with different Linux Kernel versions, which were included with some Ubuntu distributions.

Figure 2 shows the average time measured (in seconds) for FIFO Queue and UDP socket in each test. By observing our results, we can see that adding threads to read concurrently on both structures lead to a rise in execution time in all kernel versions. In other words, adding threads to applications which read concurrently from a single FIFO queue or UDP socket does not introduce any performance gain; moreover, both sources behave like an scenario where there is a single processor and reads are handled one at time. It is important to notice that newer versions show worse absolute times than older ones, but display a more thread-tolerant performance against concurrent accesses: when adding concurrent accesses, execution times in newer kernel versions does not increase as quick as it does in older ones. We can remark that, in the special case of the UDP socket, all kernels belonging to 3.X family present higher time values than 2.6 family, potentially due to Memory Accounting, feature which was introduced in version 2.6.25.

A. System Calls overhead

As we stated before, this work aims to determine the performance of network stack by itself, therefore any overhead introduced by system calls (and *libc*) should be considered in this analysis. However, the design of Linux System Calls suggests us that they are handled and executed in the same processor they are issued and, therefore, would not interfere with parallel performance. This fact leads us to consider this overhead as a constant value in each processor, since the scope of this work is analyze the scalability of network stack rather than make a detailed profiling of Linux Network API.

V. KERNEL ISOLATION

From previous section we could infer that the bottleneck is within Linux kernel, therefore we analyzed in detail how a packet is received by the kernel to check if our hypothesis is right.

A. Packet Reception Analysis

As we stated in section III, the reception process of a packet is composed of three main sub-processes: (1) Rise a HardIRQ and schedule the execution of a SoftIRQ, (2) Execution of the SoftIRQ and process the packet through network stack, and (3) Enqueue the packet in a socket queue.

The first and second are highly parallelized, being the first one asynchronously triggered by the NIC in one processor [1], [3] and the second one completely parallelized due the use of Software interruptions, allowing code *reentrancy* in the network stack [1], [3], [11], [12]. Is the third one then (the enqueuing process), where a serialization could be located, and it will be analyzed deeper due it consists, basically, in modify a shared data structure in the kernel.

When a packet has finished its journey through the network stack it has to be enqueued in the receiving or backlog queue following the next logic (represented in Figure 3) from left to right:

1) Lock the entire socket by locking the global spinlock of the structure (red instance). This does not allow other SoftIRQs to enqueue other packets concurrently.
2) Check whether the lock is being used by some user-space application or not. If it is, the packet is directly enqueued in the backlog queue (ensuring its consistency with its respective spinlock, the blue one in Figure 3) and jump to the next step. If the socket is not locked by the user, then, (1) the packet is processed through netfilter, (2) Memory Accounting statistics are updated, (3) the packet is enqueued in the receive queue (protected by with its own spinlock, green instance in Figure 3) and, (4) the system scheduler is invoked in order to wake up any asleep task waiting for data in the socket.
3) Finally, release the global socket spinlock, allowing other packet to be enqueued following this same logic.

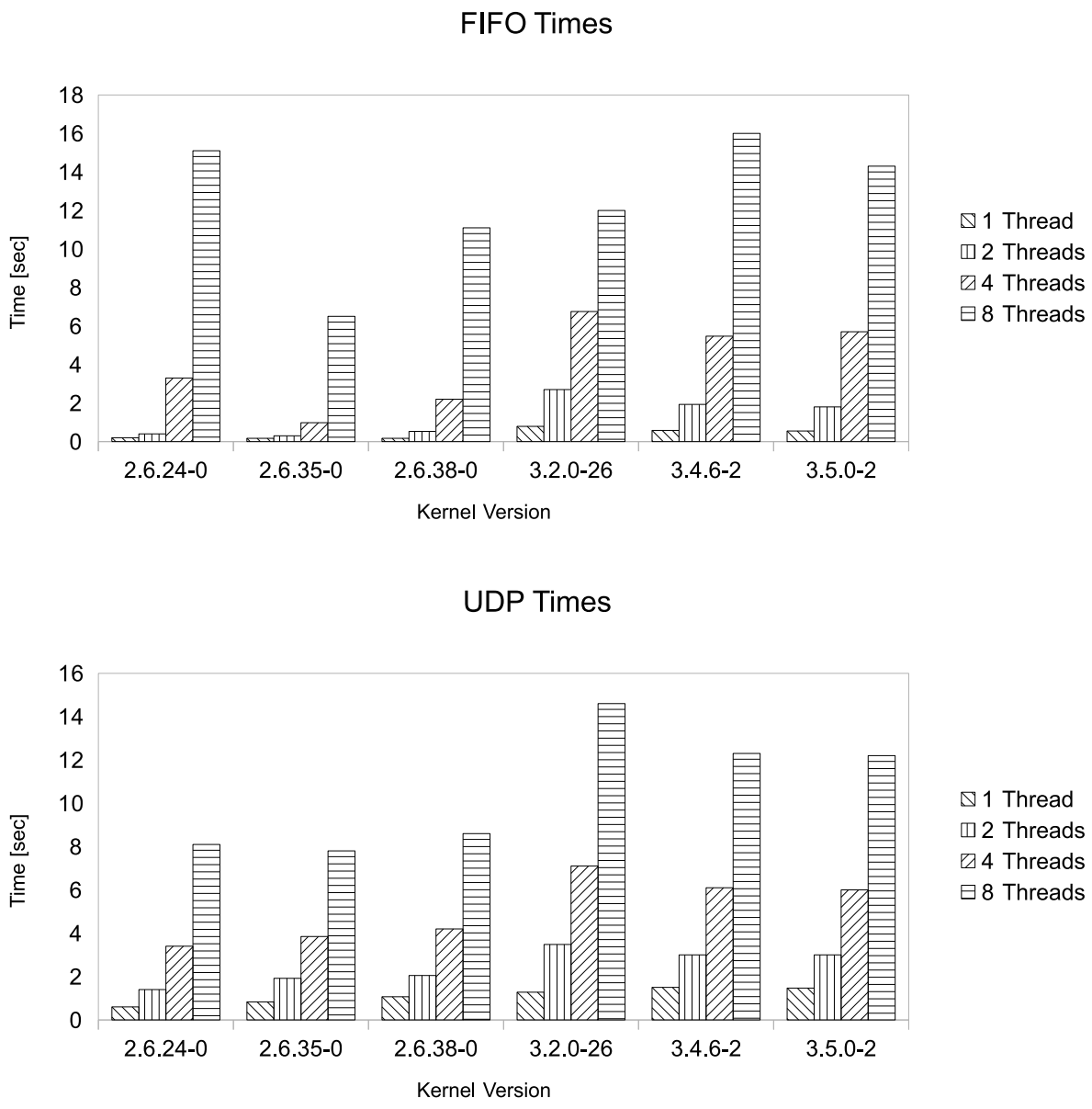

Fig. 2: Time measured in processing $500,000$ packets for $1, 2, 4, 8$ processing threads

From the other side, when the application tries to read data from the socket, it executes a similar process, which is described below and represented in Figure 3 from right to left:

1) Dequeue one or more packets from the receive queue, using the corresponding spinlock (green one).
2) Copy the information to user-space memory.
3) Release the memory used by the packet. This potentially changes the state of the socket, so two ways of locking the socket can occur: fast and slow. In both cases, the packet is unlinked from the socket, Memory Accounting statistics are updated and socket is released according to the locking path taken.

Due the process of enqueueing a packet might change the state of the socket, the kernel must avoid any race conditions with other threads by introducing two socket locking ways as stated before. The fast way occurs when no other thread is changing the socket state, and consists in holding the socket global spinlock, which deactivates *bottom half* controllers and avoid any other concurrent insertion of packets in the queue After the memory used by the packet is releases, this spinlock is released in order to allow other extractions. In contrast, the slow way occurs when other thread is handling the socket state, therefore dequeueing packets would conflict with other threads handling the socket state, especially at the moment of update Memory Accounting statistics. In this last case the calling thread will re-schedule its execution until the state lock

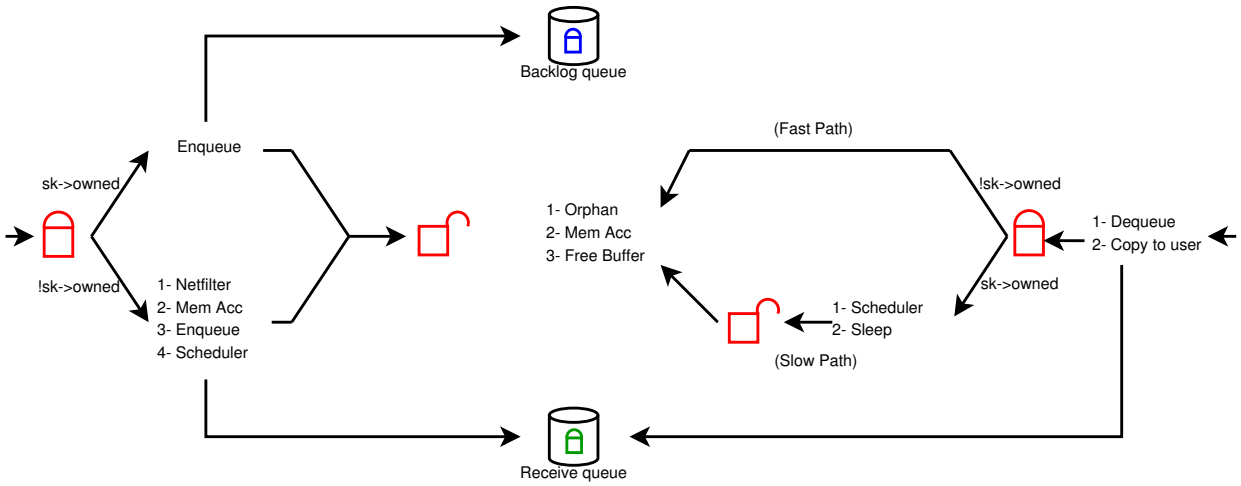

Fig. 3: Packet dequeing and enqueueing process

is released.

B. Socket Spinlocks Statistics

The analysis presented before suggests that two points might be serializing the concurrent accesses to the socket: the global spinlock of the socket or the spinlock used to maintain consistent the socket's receiving queue. This two instances are tested by running the same stressing tests for UDP sockets used in Section IV, collecting statistics about them with *lockdep* Linux kernel tool. This data collected include information about contentions, acquisition, total waiting time and total locking time.

In order to test the influence each spinlock has in the bottleneck, two scenarios will be tested: an unmodified [1] kernel and another one patched to Memory Accounting, and thus disabling socket's global spinlock. Both compilations were built using version 3.12.5 as base.

Table I[1] and Table II [1] shows held and wait average times for each spinlock in study: list and global socket spinlock. We can see that even in scenarios with one application thread, both spinlocks show wait time and hold time. A simple explanation of this is the fact that even with just one extraction from socket, the kernel might be running many SoftIRQs processing and enqueuing packets in the socket, generating contention in both synchronization primitives.

[1]Kernel code was modified just to add a lockdep class to socket global spinlock, used by lockdep to collect statistical information about it. This was done due the lack of this characteristic in the original code.

[1]For Tables I, II and III, head columns has the following meaning:

- Htime-total: Total amount of time the lock was held during the measurement.
- Acquisitions: The total number of times the spinlock was acquired.
- Htime-avg: The average time the lock was held. This value is derived from two above.
- Wtime-total: Total amount of time the lock was locked by a thread during the measurement.
- Contentions: The total number of times the spinlock contended a thread to wait for the lock.
- Wtime-avg: The time a thread had to wait for this lock.

As soon as we have more than one thread, we see a slight rise in the average time the global spinlock is held, due synchronization is required to maintain consistent the socket. Furthermore, this value seems to remain almost constant when adding even more threads. The same effect can be observed in the wait times for the same spinlock, suggesting that contention is occurring in that point. In the particular case of the queue spinlock, we can see no increment in held nor wait time for that instance when raising the number of threads, which suggests the use of that spinlock is correct.

Table III[1] shows the results for the list spinlock in the kernel where Memory Account was deactivated. We can see a slight increase in both hold and wait average times and an appreciable increment in the contentions generated by this lock. This is explained by the fact that list spinlock is now the unique synchronization point in the schema, therefore every access to the socket is contended in order to avoid a race condition in the packet queue. However, this results show that this is not the solution of the performance issue presented in this work.

In Figure 3 we present two methods to lock the socket from the user side, and each one has different effects on the control of execution. Fast path disables bottom half controller (formally SoftIRQs) by locking socket spinlock (the red one), which does not allow the enqueue of new packages neither in the backlog nor receive queue. In the case that another thread is modifying the state of the socket, the slow path is taken, where the use call is delayed until the global socket lock (the red one) is released by the other thread. Moreover, if this spinlock is disabled (by disabling Memory Account), synchronization point will remain in the receive and backlog queues (green and blue spinlocks), as showed in Table III when that feature was turned off.

This fact suggests that the enqueueing schema of the sockets is the bottleneck we are looking for: the linked list used as packet queue does not allow concurrent extractions, serializing the accesses.If concurrent accesses to a socket would like to be

TABLE I: Receive queue spinlock results, Memory Accounting enabled

Threads	Htime-total [ms]	Acquisitions	Htime-avg [ms]	Wtime-total [ms]	Contentions	Wtime-avg [ms]
1	241,739.56	1,498,456	0.16133	14.69	72	0.20403
2	491,475.11	3,000,009	0.16382	83.76	427	0.19616
4	999,715.69	6,001,903	0.16657	193.33	956	0.20223
8	2,001,099.30	12,002,304	0.16673	300.51	1,479	0.20318

TABLE II: Global socket spinlock results, Memory Accounting enabled

Threads	Htime-total [ms]	Acquisitions	Htime-avg [ms]	Wtime-total [ms]	Contentions	Wtime-avg [ms]
1	1,727,674.73	1,000,102	1.72750	122.21	222	0.55050
2	3,546,050.22	2,000,037	1.77299	1,417.54	1,235	1.14781
4	7,253,937.17	4,000,143	1.81342	3,163.35	2,576	1.22801
8	14,511,741.72	8,000,053	1.81396	5,301.16	4,281	1.23830

TABLE III: Receive queue spinlock results, Memory Accounting disabled

Threads	Htime-total [ms]	Acquisitions	Htime-avg [ms]	Wtime-total [ms]	Contentions	Wtime-avg [ms]
1	475,795.94	1,995,714	0.23841	173.13	543	0.31884
2	983,877.62	4,000,453	0.24594	677.32	1,743	0.38859
4	1,988,615.96	8,004,417	0.24844	1,147.46	3,679	0.31189
8	3,976,472.06	16,007,064	0.24842	2,429.32	8,650	0.28085

performed, Memory Accounting has to be disabled (or at least re-coded to be non-socket-blocking) and the data structure used for packets must be replaced for another one which support concurrent accesses.

VI. Multiqueue receiver network stack

We propose a naïve solution to address this problem, based on the idea of allowing multiple concurrent threads to extract packets from a single socket with the less synchronization possible.

As we stated in Section III, a Linux socket is constituted by a single receive queue, which does need synchronization against concurrent accesses to avoid pointers inconsistency in the data structure. Our proposal is a single socket with multiple receiving queues in order to allow multiple insertions and extractions at the same time. More specifically, the *multiqueue* socket will have a receive queue for each processor available in the machine, allowing threads to extract packets from the queue that belongs to the processor which runs the thread. This differs from *reuse port* flag introduced in Linux 3.9 in the way that this approach allows a single socket to use a queue per processor rather than use several sockets in a single application. Moreover, this solution has the advantage of avoid to rewrite every application in order to support reuse port.

Without going any further, we hypothesize that this approach will avoid most of the contention previously observed, with the cost of being useful only in an architecture with message-oriented parallelism. In other words, this solution will only work with servers whose queries are each self-contained in a single small packet, due our solution assigns each packet to each queue in a round-robin schema.

To test if our hypothesis was right, we implemented a multiqueue receiver network stack in the Linux kernel, available in http://github.com/niclabs/multisocket.

A simple performance test of processing $500,000$ packages of 10 bytes (as in Section IV) can be sen in Figure 4, where with a simple solution of a two-queues socket in the kernel we increase the throughput for multiple thread accesses.

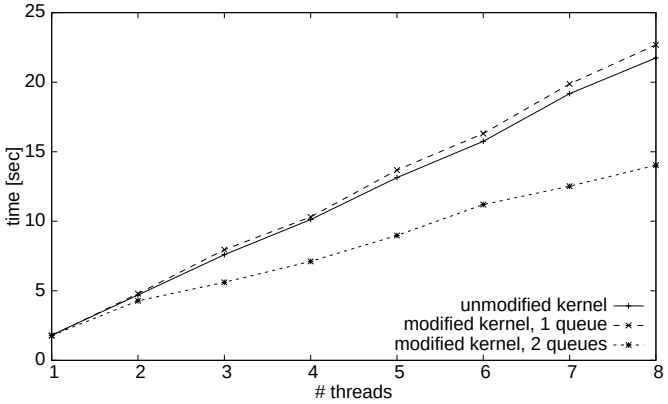

Fig. 4: Time in kernel 3.12.5 using 2 cores

VII. Conclusions and Future Work

We studied the actual performance of Linux UDP sockets, and discovered that accesses to retrieve data from them are serialized in the kernel.

Inside the kernel, we studied how the packet is received and enqueued in a socket's list and identified a possible point of failure: two spinlocks, one protecting the queue from concurrent accesses and the other used as a socket-wide lock in order to be used with Memory Accounting. We collected statistics over those spinlocks using *lockdep*, showing that the contention on the global spinlock is the main part of the problem, but deleting it will not solve the performance issues observed.

Thus, we presented a naïve queue solution for Linux kernel that achieved best performance that the actual uni-queue implementation. As future work we plan to study some other parallel data structures that could achieve better performance

than our solution and/our that can preserve the order of the packets aiming to use it in TCP-based protocols.

REFERENCES

[1] C. BENVENUTI. *Understanding Linux Network Internals*. O'Reilly Media Inc., Sebastopol, CA, USA, 2006.

[2] M. Björkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. *SIGCOMM Comput. Commun. Rev.*, 23(4):74–83, Oct. 1993.

[3] D. P. BOVET and M. CESATI. *Understanding the Linux Kernel*. O'Reilly Media Inc., Sebastopol, CA, USA, 3rd edition, 2007.

[4] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[5] E. DUMAZET. netdev - [rfc net-next-2.6] udp: Dont use lock_sock()/release_sock() in rx path. http://lists.openwall.net/netdev/2009/10/13/153, 2009.

[6] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: a new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 135–148. USENIX Association, 2012.

[7] N. Hutchinson and L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[8] T. JINMEI and P. VIXIE. Implementation and evaluation of moderate parallelism in the bind9 dns server. In *USENIX Annual Technical Conference, General Track*, pages 115–128, 2006.

[9] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[10] P. SAAB. Scaling memcached at facebook. https://www.facebook.com/note.php?note_id=39391378919, 2008.

[11] D. Schmidt and T. Suda. Measuring the performance of parallel message-based process architectures. In *INFOCOM '95*, pages 624–633 vol.2, 1995.

[12] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *USENIX Annual Technical Conference, General Track*, pages 91–96, 2006.

[13] W. Wu, M. Crawford, and M. Bowden. The performance analysis of linux networking – packet receiving. *Computer Communications*, 30(5):1044 – 1057, 2007. Advances in Computer Communications Networks.