Toroni: Lightweight Alternative To Message Brokers For Many-to-Many Publish-Subscribe Interprocess Communication

Rousko Atanasov
ratanasov@vmware.com

Kalin Tsvetkov
ktsvetkov@vmware.com

ABSTRACT
In RADIO 2021 some authors presented the paper Brokerless Many-to-Many Publish-Subscribe Interprocess Communication [2]. The latter runs on top of a simple reliable totally ordered many-to-many multicast protocol on a single node that uses UDP as data plane and shared memory as control plane. In this paper, we set our goal as improving the performance of their solution. We have extended the authors' research and propose a more lightweight and efficient architecture. We build and evaluate a prototype of the proposed protocol stack as a library named Toroni. We believe our design can be applied in products willing to isolate features in separate processes on the same machine.

General Terms
Algorithms, Design, Reliability, Experimentation.

Keywords
Distributed Systems, Publish/Subscribe Messaging, Reliable, Total Order, Multi-Source, Multicast, Broker-less.

1. INTRODUCTION
In this paper, our goal is to extend the research in [2] identifying a more lightweight and efficient solution. As in [2], the premise of this paper does not imply that [hidden] should be broken up into a set of processes but only that if such conclusion is made, the problem discussed in [2] and this paper becomes relevant. Then we present our new broker-less design (see Section 2 for our motivation) for an inter-process Channel [1] and a prototype [8]. By doing so, we hope to provide some of the infrastructure to move [hidden] modules into separate processes and improve availability of loosely coupled functionalities. This will also allow faster implementation of new features without destabilizing the existing ones. Our approach is not specific to [hidden] and can be applied to Linux as well. To achieve this goal, we divide our effort into the following objectives:

- Evaluate existing mechanisms for inter-process communication (IPC).
- Propose a simpler and more efficient design which meets the requirements.
- Implement a prototype and evaluate it.

2. BACKGROUND
As the problem continues to remain the same, we advise the reader to refer to its description in [2] but suffice it to say it deals with migrating the existing [hidden] daemon intra-process asynchronous messaging system to inter-process with minimum intrusion and almost no explicit changes to the existing codebase. In this paper, we use the terms channel, channel name and topic interchangeably. By reader and writer we will refer to the process which runs a ChannelManager instance. By ChannelReader and ChannelWriter we will mean instances created by ChannelManager.

The existing intra-process design of Channel has a central entity, the ChannelManager, which decouples ChannelReaders and ChannelWriters. A channel is a queue that stores messages sent from channel writers. Channels have names with tree-like hierarchical relationship, allowing readers to subscribe for a channel and all its children, and writers to post to a channel and all its children. All readers of a channel see the posted messages in the same order and only once.

There are some additional challenges posed by becoming inter-process:

- Brokerless. The Channel interface can be implemented using a broker-based approach. However, a broker rapidly becomes a bottleneck and creates a new risk to manage. It creates more moving pieces, more complexity, and more things to break. If the broker isn't running, the communication can't happen [2].
- Many-to-Many. A channel has runtime dynamic set of writers and readers.
- Total Order. All readers of a channel must see the matching messages in the same order.
- Reliability. A reader process may miss a message. The reader should know if this happens and handle it.
- Termination safety. A process may crash or be terminated (SIGKILL). This should not break the remaining communicating processes.

A thorough investigation of existing solutions is done in [2] and summarized in Table 1.

Table 1. Summary of existing solutions and their applicability

	Broker-less	Many-to-Many	Total Order	Reliable	Terminate-Safe
Message queue	yes	no			
Shm+Disruptor	yes	yes	yes	yes	no
UDP multicast	yes	yes	no	no	yes
UDS	yes	no	yes	yes	yes
PGM	yes	yes	no	yes	yes
SRM			no		yes
0MQ pub/sub	yes	yes	no	no	yes
0MQ Xpub/Xsub	no				yes
0MQ pgm/epgm	yes	yes	no		yes
Aeron	no		no	yes	yes

As concluded in [2], none of them, when used in isolation, meets our requirements. Owning the [hidden] stack end-to-end, augmenting the [hidden] behavior with Channel (or as kernel module) may be feasible. However, this will create additional friction for adoption and will be specific to [hidden] and not

applicable to Linux-based appliances. Therefore, we have opted for a user-space approach.

3. SYSTEM ARCHITECTURE

In this section we will demonstrate how using the underlying technologies from [2] we can produce a simpler and more efficient solution. We propose a stack of protocols that run on top of unreliable multicast and shared memory to produce a reliable totally ordered many to many publish/subscribe system. It is packaged as a library named Toroni that is used by the communicating processes.

In [2] it is already identified that using shared memory as data-plane requires to address two main issues:

- The first one is to ensure termination safety when a process dies in-between acquiring and releasing an OS-mutex. With a regular inter-process mutex, if an updating process holding it dies before releasing it, other processes will block forever trying to acquire it.
- The second one is to identify such a data structure that is recoverable when left in inconsistent state by the crashed process.

3.1 Robust Futex

The concepts of futex and robust futex are described by Ingo Molnar in [3].

A futex is an optimized lock that in the non-contended case can be acquired/released from userspace without entering the kernel.

A robust futex can handle process crash while holding a lock. If such a lock is also shared with some other process, then they need to be notified that the last owner of the lock exited in some irregular way. In such case, they can recover the possibly corrupt shared state and mark the futex as healthy.

At the heart of a robust futex is a per-thread private list of robust locks that userspace is holding which is registered with the kernel. If a process terminates in some irregular way, the kernel walks the list and marks all locks that are owned by this thread with the FUTEX_OWNER_DIED bit and wakes any waiters. Additional glibc support is also needed to properly handle the case when a process crashes after acquiring the lock, but just before adding it to the list.

[hidden] supports robust futex and its glibc has the proper list support [6], [7].

Robust futexes take us one step closer to using shared memory as data plane while maintaining termination safety. The other required thing is a data structure that when left in inconsistent state, can be recovered.

3.2 Reliable Message Protocol (RMP)

The core protocol from the proposed stack is the Reliable Message Protocol (RMP). It provides reliable totally ordered many-to-many protocol within a single node for messages with arbitrary length by using shared memory, as data plane, and unreliable multicast, as control plane. The data unit of RMP is a message. Its header is comprised of a type and length fields (Figure 1). RMP uses a segment of shared memory that is pre-allocated and pre-initialized with robust data structures protected by robust futex.

3.2.1 Byte Ring Buffer

The ByteRingBufer is an infinite stream of arbitrary length messages mapped onto a ring buffer of fixed length in shared memory (Figure 1).

Each message starts with a header with the message length and type. The stream end is an atomic monotonically increasing uint64 counter (freePos) in shared memory. It is mapped (via modulo) to an index in the ring buffer. A message is appended to the message stream by (1) copying it to the shared memory and (2) increasing freePos (commit). This operation is protected by a robust futex to honor the single-writer principle which key for being race-free. If a writer has crashed before updating freePos, its message is not seen by readers and is overwritten by the following writer. A following writer that acquires the robust futex doesn't need to recover the ring buffer.

If a message cannot fit in the ring buffer starting at the current index, a padding message is added to the end of the ring buffer and then the real message at the front. Padding messages are ignored when reading.

Each reader has a stream position and when reading, gets the messages from that position up to the stream end. Reader's access to the ring buffer is wait-free. It reads a message from the stream by copying it from shared memory. After copying it, it checks if the stream end position has overrun the reader's stream position (in terms of index in the ring buffer). If it has, the reader pessimistically assumes (message append may have happened after copy-reading but before checking) it has missed an unread message. If it hasn't, then there is 100% certainty that no message has been lost. Naturally, all readers can see all messages from the ring buffer in the same order.

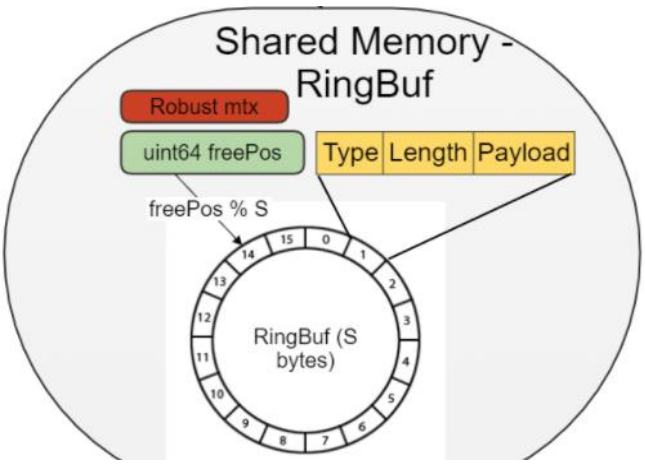

Figure 1. Byte Ring Buffer

3.2.2 Backpressure

Writers append to the infinite message stream and readers catch up with writers. The message stream is mapped to a circular buffer of fixed size. If a reader is slow to catch up, the buffer will be overwritten by a writer and the reader will miss a message (Figure 2). We have confirmed experimentally that without backpressure a writer can outpace a reader even when both run as fast as they can. In general, increasing the size of the ring buffer, reduces the chance for backpressure. Writers could detect this condition and have a policy to let readers catch up.

Figure 2. Backpressure

Different backoff protocols can be used. The currently used protocol, is to have the writer first check for backpressure, then sleep for some time and finally retry writing the message to the ring buffer without backpressure. Another protocol could be to use exponential backoff before writing the message. Regardless of what protocol is used, writers must not wait indefinitely for readers, as they may have crashed.

Readers with their stream position that are overwritten (expired) cannot create more backpressure as the message stream advances.

3.2.3 Reader Info

The reader keeps its stream position in its process and only copies it to the shared memory for the purpose of backpressure. Thus, receive-wise, one reader cannot corrupt the stream position of another reader. There is a pre-defined maximal number of readers allowed in the shared memory (Figure 3). Each is assigned a stream position that is stored in a reader-info shared memory segment. This way a writer can check if it is going to overwrite any reader and if so, create backpressure. Each reader stream position is also protected by a robust futex. It is acquired via try-lock when a reader starts and released when done. While acquired, it cannot be assigned to another reader. Should a reader crash, other readers can re-acquire it. There is no process whose crash may affect the remaining communicating processes.

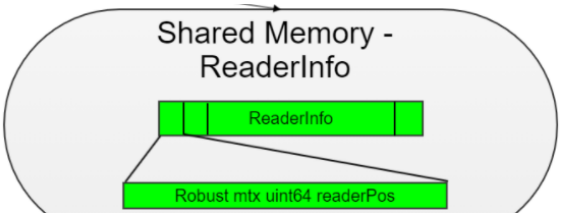

Figure 3. Reader Info

3.2.4 Notification

Without use of inter-process notification, there is a tradeoff between message latency and CPU usage. There must be a way to notify a waiting reader that a new message has been added to the stream.

An obvious candidate are inter-process conditional variables. POSIX supports such but they are not robust regarding process crash like the robust futex and this is not considered a bug [4].

Another candidate is eventfd. It creates a kernel object for event notification and returns file descriptor. The object contains an unsigned 64-bit integer. An eventfd read operation decrements the counter by 1 or blocks if it is 0. An eventfd write operation adds a value to the counter or blocks if it reaches the maximum value [5]. However, an eventfd is not suitable for notify_all semantics because a reader process A can decrement the eventfd counter twice leading to a reader process B missing a notification. Additionally, sharing the file descriptor among dynamic set of unrelated processes may be tricky.

Our architecture uses UDP multicast (Figure 4). To notify readers, writers send a dummy UDP message to a multicast address after freePos has been updated. One notification can be sent for several messages. All necessary information is in shared memory, and none is in the UDP message. Readers are insensitive to what data is in the UDP message. Thus, readers can wait for a notification and be CPU-friendly without sacrificing latency. False or missed notifications are not harmful - a reader will identify when there is nothing to read.

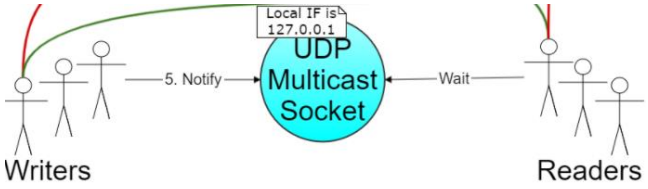

Figure 4. UDP Notification

Using UDP also allows efficient thread usage as it is pollable. The reader process can have a single thread that polls many file descriptors.

Our choice to use UDP instead of signals for notification is because using signals seems tricky in the following ways: First, a signal handler must call only async-signal-safe functions, and the signal handler itself must be reentrant with respect to global state in the program. A function is async-signal-safe either because it is reentrant or because it is atomic with respect to signals. Second, having a process group of readers signaled by writers, requires that either the reader processes are related (a too hard requirement), or a process group is explicitly created. It may be tricky to negotiate this process group between readers and writers while none of them is running.

3.2.5 Security

Security per channel (i.e., only processes with the right set of permissions can do operations on a channel) is outside the scope of this paper.

There is no requirement that a reader knows the source of the message, i.e., whether it was component A posting to channel C or component B posting to channel C.

To guarantee that whoever wrote the message, had the proper permissions, we can set permissions on the shared memory segments and place the ring buffer and reader info in separate segments (Figure 5).

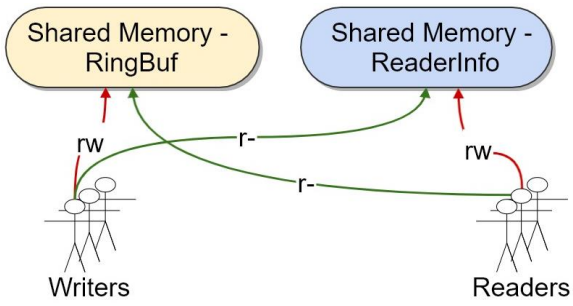

Figure 5. Security

For the ByteRingBuffer segment, only writers can post a message and readers can only read it. Permission-wise writers have read-write access and readers have read access.

For the ReaderInfo segment, writers can only read it to check for backpressure and readers can read-write it. The source of truth for a reader's position is its in-process field, which is copied onto the ReaderInfo segment for the purpose of backpressure. This ensures

that a malicious agent cannot trick a reader to skip messages or read the same messages again. In other words, a malicious reader is only able to create/avoid creating backpressure e.g., by writing to its own or others' ReaderInfo segments and is not able to cause message loss. A reader can securely and reliably detect message loss.

The POSIX Access Control Lists are suitable for implementing this security model (Table 2).

Table 2. Reader and Writer ACLs

	RingBuffer	ReaderInfo
Writers Group	rw	r
Readers Group	r	rw

3.3 Topic Protocol (TP)

The other protocol that complements the proposed stack is the Topic Protocol (TP). TP runs on top of RMP and provides the desired ChannelManager pub/sub protocol carrying the (topic, postToDescendants, message) tuples, with topic filtering on the reader-end (see Figure 6). As an optimization, a reader can copy only the topic of the message, check for subscription and if none is found, just increase its in-proc stream position without copying the rest of the message.

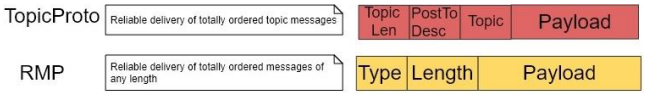

Figure 6. Protocol Stack

Message serialization/deserialization onto TP is outside the scope of this paper. However, in [hidden]'s case it's not rare when the message payload is none/very small/must be very small. Today this may not be the case as there is no incentive for this because passing a reference within a process is cheap.

4. EXPERIMENTS AND EVALUATION

4.1 Integration

In this experiment we integrated Toroni with the [hidden] library, created a standalone application that listens for all topics posted by [hidden] and prints them on the console. We ran two reader instances on [hidden] and were able to see messages.

4.2 Intra-Pod Communication

In this experiment our goal was to explore intra-Pod communication using Toroni. As a proof of concept, we conducted a test with reader and writers in separate containers in the same Kubernetes pod. The pod provides shared IPC namespace for the shared memory segment (Figure 7).

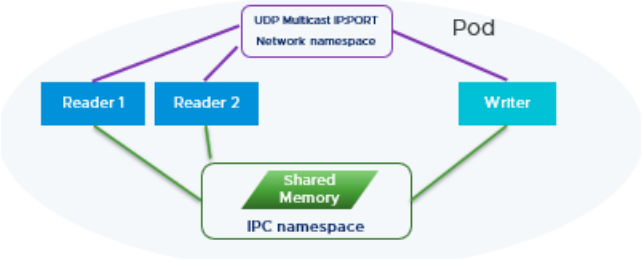

Figure 7. Intra-Pod Communication

Being broker-less, Toroni has minimal resource footprint. This makes it a viable option for a message bus on resource constrained edge devices.

4.3 RMP vs TCP on ESXi

In this experiment we evaluated the throughput of Toroni RMP with 1 reader and 1 writer and compared it to TCP. The tests were run on a physical ESXi. Figure 8 shows the ratio (TCP/Toroni) of time to receive 100K messages depending on message size. We can see that our approach is up to 800% faster than TCP loopback.

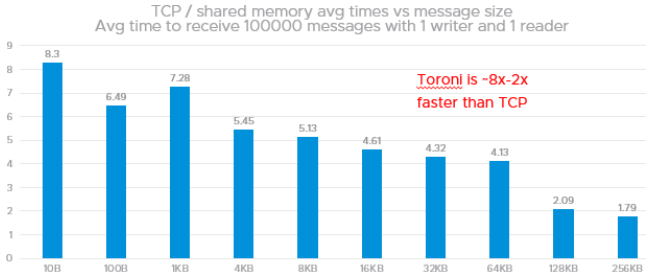

Figure 8. RMP vs TCP and its dependence on message size

4.4 Topic Protocol Performance on Ubuntu

We gathered different metrics with the purpose to measure the end-to-end performance of the system. We used a protocol implementation which is not tuned for performance [8]. We ran our experiments on an Ubuntu 18.04 virtual machine with 8GB RAM and 16 vCPUs on 2 sockets running on a Dell PowerEdge R630, having 20 Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz on 2 sockets and 256 GB RAM. A ByteRingBuffer of 4MB was used. Using a larger buffer will decrease the chance for backpressure.

The purpose of the first metric is to measure the throughput and its dependence on the number of writer and reader nodes (see Table 3). Please, note that by reader and writer nodes we mean processes that run ChannelManager and not the ChannelReader and ChannelWriter instances created by it. Each writer node is producing a burst of 20000 few byte messages. We measured throughput as the number of received messages from all writers divided by the time interval between the first and the last message. The values are average between all readers across 5 runs of the experiment. As it can be seen from the table, with 7 writers, 16 readers and 140000 messages to be received by each, the average throughput is 308000 messages/sec.

Table 3. Throughput of messages per second (K msg/sec) and its dependence on number of writers and readers

W 1-7 R 1-16	1	2	3	4	5	6	7
1	1429	1275	1380	1419	1256	1268	1120
4	1222	1133	1029	963	947	903	854
8	999	834	816	786	778	745	731
16	374	366	328	339	356	326	308

4.4.1 Performance Tuning

The TP protocol used so far sends notification after using 1% of the ring buffer and after the writer thread ends. Our benchmarks show that sending a UDP notification may be a lot costlier than writing a small message to the ring buffer. Therefore, it is a low hanging fruit to reduce the number of UDP notifications by integrating them with backpressure. The writer thread will send a notification on backpressure, then invoke the backpressure policy, in addition to when it exits/waits.

Another observation is that it is important how the reader updates its ReaderInfo slot in shared memory with its in-process local position. The current TP protocol, updates shared memory after

reading every message. However, the ReaderInfo memory layout is likely to suffer from false sharing. False sharing is a well-known performance issue on SMP systems, where each processor has a local cache and occurs when threads on different processors

modify variables that reside on the same cache line. Since the memory system must guarantee cache coherence, this hurts performance.

Removing some pack pragmas and addressing the above considerations yields a significant improvement in throughput. The values shown in Table 4 are average among all readers across 500 runs.

Table 4. Improved throughput of messages per second (Million msg/sec) and its dependence on number of writers and readers

W 1-7 R 1-16	1	2	3	4	5	6	7
1	21.2	19.7	14.3	12.8	11.5	10.5	9.57
4	21.2	19.4	11.4	8.62	7.29	7.14	7.03
8	20.9	17.0	8.44	6.72	6.64	6.41	6.25
16	19.8	15.9	7.06	5.85	5.68	5.68	5.46

We can see average improvement of 1000 X compared to the results in [2].

We conducted series of experiments to measure the latency and its dependence on the number of writer and reader nodes (see Table 5). Each writer posts burst of 10 messages. It puts a timestamp in the message. Upon receipt, the reader evaluates the latency against the message timestamp. The values shown in table are average among all readers across 500 runs. As it can be seen the average latency for the 7-16 case is 98 µsec. We can see average improvement of 190 X of the results in [2]. Moreover, compared to the polling design in [2], a Toroni idle reader has zero CPU usage thanks to the notification between writers and readers.

Table 5. Latency in microseconds and its dependence on number of writers and readers

W 1-7 R 1-16	1	2	3	4	5	6	7
1	53	46	43	43	42	43	42
4	66	57	54	53	53	53	53
8	115	92	82	76	75	71	69
16	200	147	123	113	107	101	98

4.4.2 Performance Analysis

To analyze the results in Table 4 we need to look at more metrics available in Table 4* (see Appendix). For a fixed number of writers, throughput decreases as the number of readers increases because it takes writers more time to write their messages (WT). It is comprised of increasing time to write to the ring buffer (WW) plus increasing time for UDP notifications (WN) (looks like a scaling issue with UDP multicast) and WW outweighs WN. Moreover, this may increase the number of reader wakeups (R) which is expensive. Backpressures (B) are not observed for less than 7 writers, and more backpressures mean more notifications (see 7 writers). For a fixed number of readers, throughput decreases as the number of writers increases because more writers mean more contention for the ring buffer, meaning more time to write their messages. This also increases reader wakeups (R) which is expensive. We observe batching - the number of reader wakeups is less than the number of writers (for more than 1 writer). This implies that the reader picks up messages from other writers before going back to sleep waiting for their notification, and therefore more writers means fewer cases of the readers finding the ring buffer empty. Figure 9 shows how for 7 writers throughput decreases as the number of readers increases.

To analyze the results in Table 5 we need to look at more metrics available in Table 5* (see Appendix). For a fixed number of writers, more readers decrease performance because it takes writers more time to write their messages (WT). It is comprised of increasing time to write to the ring buffer (WW) plus increasing time for UDP notifications (WN) and WN dominates. Backpressures (B) are not observed. Batching i.e., R less than number of writers, is observed. For fixed number of readers, more writers mean less latency because of batching. Figure 9 shows how for 7 writers, latency increases with the number of readers.

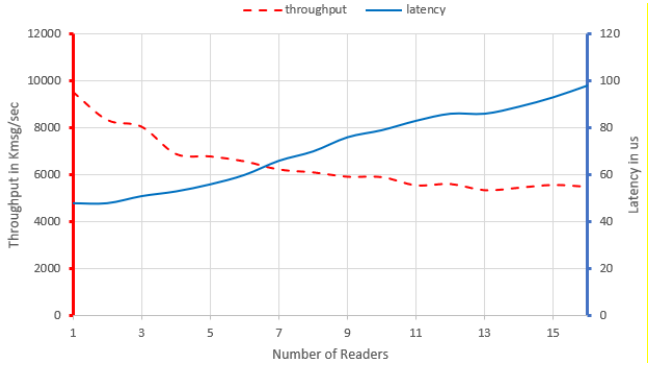

Figure 9. Throughput and latency of 7 writers and its dependence on the number of readers.

Our objective was to design a broker-less reliable totally ordered many-to-many publish/subscribe system on a single node. The current performance indicates usable results, and we believe they can be improved if necessary.

We were unable to identify existing solutions that meet our requirements in order to compare Toroni with their performance.

4.4.3 Stress Test

We conducted a stress test of transferring 70MB using a 64KB ring buffer (20K 500-byte messages from 7 writers to 16 readers, 5ms backpressure sleep). We ran it 10 times, no readers expired and obtained average throughput of 23.45 Kmsg/sec and WT=5653.304, WW=5653.152, WN=0.153, E=0, N=162.286, B=1129.000, R=1136.000 (see Appendix for notation).

5. CONCLUSION

In this paper, we continued the authors' research in [2] for a broker-less reliable many-to-many publish/subscribe inter-process communication on a single node. Unlike [2] which uses shared memory as control plane, our current approach does the opposite. To accomplish this, robust futex and a suitable termination safe data structure, such as the ByteRingBuffer, are used. Placing ByteRingBuffer in a segment shared memory segment allows for a simpler protocol stack than [2].

Our new approach is CPU friendly unlike [2]. To achieve CPU friendliness, we have considered several notification techniques and settled down on UDP mulitcast.

We have tested Toroni on Ubuntu and it demonstrated 1000X better throughput and 190X better latency than the results achieved in [2].

Further, we integrated Toroni with the [hidden] library and ran it on ESXi. We observed that, on ESXi, Toroni's RMP is up to 8X faster than TCP.

We believe our approach provides a simpler alternative to the complexity of using a broker-based messaging system within Linux-based operating systems.

6. REFERENCES

[1] *hidden*

[2] Atanasov, R., Tsvetkov, K., Bambaldokova, V. 2021. *Brokerless Many-to-Many Publish-Subscribe Interprocess Communication.* In Proceedings of the VMware RADIO Conference 2021.

[3] Molnar, I., *A description of what robust futexes are.* https://www.kernel.org/doc/Documentation/robust-futexes.txt

[4] *Deadlock in 2.25 pthread_cond_broadcast after process abort.* https://sourceware.org/bugzilla/show_bug.cgi?id=21422

[5] *eventfd*, https://man7.org/linux/man-pages/man2/eventfd.2.html

[6] *Robust futex / pthread_mutex support*. https://bugzilla.eng.vmware.com/show_bug.cgi?id=1217653.

[7] *[CAT][main-stage]pthread test failed when initialize the mutex attributes.* https://bugzilla.eng.vmware.com/show_bug.cgi?id=1522848

[8] *Prototype source code.* https://gitlab.eng.vmware.com/ratanasov/shmem-channel/-/tree/shmonly/copyHandler/shmonly

Appendix

WT - Average time per writer in ms to write all messages including notifications

WW - Average time per writer in ms for pure writing excluding notifications

WN - Average time per notification in ms

N - Average number of sent notifications per writer

B - Average number of backpressure events

R - Average number of reader runs (i.e wake-ups) per reader

Table 4*. Improved throughput of messages per second (Kmsg/sec) and its dependence on number of writers and readers

W 1-7 R 1-16	1	2	3	4	5	6	7
1	21285.802 WT=1.484 WW= 1.459 WN=0.025 N=1.0 B=0.0 R=1.000	19773.438 WT=9.126 WW=9.107 WN=0.019 N=1.0 B=0.0 R=1.028	14350.078 WT=13.499 WW=13.480 WN=0.018 N=1.0 B=0.0 R=1.342	12806.475 WT=22.199 WW=22.178 WN=0.020 N=1.0 B=0.0 R=1.764	11572.854 WT=29.909 WW=29.885 WN=0.023 N=1.0 B=0.0 R=2.410	10563.718 WT=44.568 WW=44.547 WN=0.021 N=1.0 B=0.0 R=3.006	9566.925 WT=57.585 WW=57.562 WN=0.023 N=1.017 B=0.120 R=4.282
4	21222.900 WT=1.658 WW=1.613 WN=0.045 N=1.0 B=0.0 R=1.000	19446.376 WT=12.027 WW=11.996 WN=0.031 N=1.0 B=0.0 R=1.010	11439.166 WT=15.191 WW=15.147 WN=0.043 N=1.0 B=0.0 R=1.401	8619.041 WT=25.789 WW=25.748 WN=0.041 N=1.0 B=0.0 R=2.097	7294.423 WT=36.482 WW=36.419 WN=0.062 N=1.0 B=0.0 R=2.841	7139.089 WT=52.892 WW=52.814 WN=0.079 N=1.0 B=0.0 R=3.479	7027.516 WT=68.663 WW=68.577 WN=0.087 N=1.014 B=0.100 R=4.730
8	20985.811 WT=1.912 WW=1.795 WN=0.117 N=1.0 B=0.0 R=1.000	17049.233 WT=13.630 WW=13.566 WN=0.064 N=1.0 B=0.0 R=1.020	8439.921 WT=18.588 WW=18.487 WN=0.101 N=1.0 B=0.0 R=1.621	6722.262 WT=29.942 WW=29.835 WN=0.107 N=1.0 B=0.0 R=2.441	6639.481 WT=43.117 WW=42.986 WN=0.131 N=1.0 B=0.0 R=2.988	6408.012 WT=61.224 WW=61.078 WN=0.146 N=1.0 B=0.0 R=3.373	6247.271 WT=78.966 WW=78.715 WN=0.251 N=1.026 B=0.180 R=5.293
16	19800.658 WT=2.627 WW=2.159 WN=0.468 N=1.0 B=0.0 R=1.000	15950.822 WT=16.309 WW=15.297 WN=1.012 N=1.0 B=0.0 R=1.070	7064.714 WT=23.388 WW 22.106 WN=1.282 N=1.0 B=0.0 R=1.929	5845.759 WT=36.718 WW=35.388 WN=1.329 N=1.0 B=0.0 R=2.770	5679.202 WT=52.117 WW=50.809 WN=1.308 N=1.0 B=0.0 R=3.514	5684.049 WT=71.612 WW=70.352 WN=1.260 N=1.0 B=0.0 R=4.338	5458.724 WT=88.844 WW=87.487 WN=1.357 N=1.037 B=0.260 R=5.945

Table 5*. Latency in milliseconds and its dependence on number of writers and readers

W 1-7 R 1-16	1	2	3	4	5	6	7
1	0.053 WT=0.028 WW= 0.005 WN=0.024 N=1.0 B=0.0 R=1.000	0.046 WT=0.027 WW= 0.005 WN=0.022 N=1.0 B=0.0 R=1.882	0.043 WT=0.027 WW= 0.005 WN=0.022 N=1.0 B=0.0 R=2.938	0.043 WT=0.027 WW= 0.006 WN=0.022 N=1.0 B=0.0 R=3.926	0.042 WT=0.027 WW= 0.005 WN=0.022 N=1.0 B=0.0 R=4.940	0.043 WT=0.028 WW= 0.006 WN=0.022 N=1.0 B=0.0 R=5.908	0.042 WT=0.028 WW= 0.006 WN=0.022 N=1.0 B=0.0 R=6.948
4	0.066 WT=0.048 WW= 0.005 WN=0.043 N=1.0 B=0.0 R=1.000	0.057 WT=0.045 WW= 0.005 WN=0.040 N=1.0 B=0.0 R=1.873	0.054 WT=0.045 WW= 0.006 WN=0.040 N=1.0 B=0.0 R=2.916	0.053 WT=0.046 WW= 0.006 WN=0.040 N=1.0 B=0.0 R=3.902	0.053 WT=0.046 WW= 0.006 WN=0.041 N=1.0 B=0.0 R=4.902	0.053 WT=0.046 WW= 0.006 WN=0.040 N=1.0 B=0.0 R=5.891	0.053 WT=0.046 WW= 0.006 WN=0.040 N=1.0 B=0.0 R=6.856
8	0.115 WT=0.115 WW= 0.005 WN=0.109 N=1.0 B=0.0 R=1.000	0.092 WT=0.078 WW= 0.006 WN=0.072 N=1.0 B=0.0 R=1.869	0.082 WT=0.076 WW= 0.006 WN=0.070 N=1.0 B=0.0 R=2.877	0.076 WT=0.074 WW= 0.006 WN=0.069 N=1.0 B=0.0 R=3.857	0.075 WT=0.078 WW= 0.006 WN=0.071 N=1.0 B=0.0 R=4.839	0.071 WT=0.074 WW= 0.006 WN=0.068 N=1.0 B=0.0 R=5.855	0.069 WT=0.078 WW= 0.006 WN=0.072 N=1.0 B=0.0 R=6.859
16	0.200 WT=0.208 WW= 0.006 WN=0.202 N=1.0 B=0.0 R=1.000	0.147 WT=0.187 WW= 0.006 WN=0.180 N=1.0 B=0.0 R=1.805	0.123 WT=0.187 WW= 0.007 WN=0.180 N=1.0 B=0.0 R=2.721	0.113 WT=0.176 WW= 0.007 WN=0.169 N=1.0 B=0.0 R=3.665	0.107 WT=0.176 WW= 0.007 WN=0.169 N=1.0 B=0.0 R=4.639	0.101 WT=0.170 WW= 0.007 WN=0.163 N=1.0 B=0.0 R=5.727	0.098 WT=0.168 WW= 0.007 WN=0.161 N=1.0 B=0.0 R=6.708