REICoM: Robust and Efficient Inter-core Communications on Manycore Machines

Pierre-Louis Aublin
Grenoble University

Sonia Ben Mokhtar
CNRS - LIRIS

Gilles Muller
INRIA

Vivien Quéma
Grenoble INP

***Abstract*—Manycore machines are becoming an alternative to physically distributed systems. The software running on these machines more and more resembles distributed, message-passing applications. Consequently, these machines require robust and efficient inter-core communication mechanisms. In this paper, we study ten communication mechanisms that are considered state-of-the-art. We show that only two communication mechanisms are robust, but that, unfortunately, they are not efficient. We do thus propose REICoM, a new inter-core communication mechanism that is both robust and efficient. Using two macro-benchmarks (a consensus and a snapshot protocol), we show that REICoM consistently improves the performance over existing mechanisms.**

***Keywords*-Manycore machines; inter-core communication; fault-tolerance; operating systems**

I. Introduction

Modern computers have an increasing number of cores. They become an alternative to physically distributed systems as they are economical and consume less energy than their distributed counterparts for running servers applications. The distribution of these manycore systems inherently calls for the development of fault-tolerance mechanisms, e.g., replication protocols, checkpointing protocols, broadcasting protocols, etc. in order to deal with software and hardware failures. As an example, several recent studies have proposed multicore versions of fault-tolerance protocols: PaxosInside [8] is a consensus protocol in the lineage of the Paxos protocol for manycore machines, and Chun et al. have proposed to adapt Byzantine Fault Tolerant protocols to manycore machines [4]. Moreover, the Barrelfish operating system [2] runs, on top of a novel communication mechanism specially devised for efficient point-to-point communication on manycore machines, a two phase commit replication algorithm to maintain a consistent state of the system on the different cores. Finally, fault-tolerant protocols often require checkpointing [10] and snapshot algorithms [14] in order to recover from a failure.

These fault-tolerant protocols require communication mechanisms that are both robust and efficient. Regarding robustness, we expect from a mechanism that it protects message integrity against faulty user-level processes and that it properly handles the crash of either senders or receivers. This is the case of most kernel-level mechanisms (e.g., Pipes) contrary to user-level ones (e.g., the message-passing mechanism of the Barrelfish OS). Regarding efficiency, we expect from a mechanism that it provides an adequate one-to-many communication primitive. That is to say, it has to minimize the number of message copies and of system calls needed for a one-to-many message exchange. However, all the state-of-the-art communication mechanisms we have studied fail in providing efficient one-to-many communication primitives, as either the number of memory copies (e.g., Unix Domain sockets) or of system calls (e.g., IPC Message Queues) is a function of the number of receivers.

We present in this paper the Robust and Efficient Inter-core Communication Mechanism (REICoM). To the best of our knowledge, it is the first kernel-level inter-core communication mechanism that is robust and efficient. The robustness of REICoM is characterized by the following properties: (1) the memory zone in which the messages are written is not alterable by user-level processes, as it is protected by the kernel and (2) the crash of the senders and the receivers is properly handled as it listens to process crash events sent by the kernel. The efficiency of REICoM is due to the fact that it minimizes both the number of system calls and the number of message copies required to send a message to a set of receivers. Specifically, REICoM requires only one message copy and one system call for sending a message to a set of N receivers. REICoM is not intended to replace existing mechanisms for applications that require only one-to-one communications. Instead, REICoM is optimized for one-to-many communications. Indeed, we demonstrate in our performance evaluation that even for applications that rely on a minority of one-to-many communications, using REICoM makes a non-negligible difference in performance. For instance, with 24 nodes, the snapshot

algorithm proposed in [14], which we implemented, requires one one-to-many communication and twenty-three one-to-one communications. Using REICoM, this protocol is twice as fast as using Pipes, which is the most efficient state-of-the-art communication mechanism that provides the same level of robustness.

We compare analytically the robustness of REICoM and experimentally its performance, to ten state-of-the art communication mechanisms. Our analysis shows that only two of the ten mechanisms, i.e., TCP sockets and Pipes, offer the same level of robustness as REICoM. Finally, we evaluate the performance of the different communication mechanisms on a 24-core machine, using PaxosInside [8] and the snapshot protocol of Manivannan et al. [14]. These applications use a mix of one-to-one and one-to-many communications. Results show that REICoM systematically outperforms all state-of-the-art mechanisms. More precisely, REICoM outperforms the two robust mechanisms, i.e., TCP and Pipes by up to 920% and 300% for TCP and 190% and 140% for Pipes in the two benchmarks, respectively.

The remaining of the paper is organized as follows: Our system model is detailed in Section II. In Section III, we describe and analyze the existing communication mechanisms that can be used for inter-core communications. We conclude that none of the existing mechanisms is both robust and efficient for inter-core communications. We do thus present REICoM, in Section IV. We then compare the performance achieved by REICoM to that achieved by state-of-the-art protocols in Section V. We finally discuss related work in Section VI, before concluding the paper in Section VII.

II. System model

We consider a manycore machine on which message-passing applications are run. The communications between the different cores are assumed to be eventually synchronous. Namely, synchronous periods, during which there is an unknown bound Δ after which a sent message is delivered, occur infinitely often, so that the system can eventually make progress. This model captures the fact that a core may be slow or that the inter-core links may become saturated. Moreover, messages may be lost, duplicated or delivered out-of-order.

We use the terms faults, errors and failures as defined in [1]. We assume that the fault model is software arbitrary faults. For instance, a process may crash, corrupt its memory area, or become unresponsive. Finally, the machine and the kernel compose our Trusted Computing Base (TCB): they are supposed to be correct and resilient to faults.

III. Background

In this section, we analytically compare ten inter-core communication mechanisms with respect to their robustness and their efficiency. The seven first mechanisms are kernel-level mechanisms provided by the traditional Linux/Unix operating systems. Some of these mechanisms have been devised for communication between processes residing on the same host, namely Unix domain sockets, Pipes, Pipes+`vmsplice()`, IPC message queues and POSIX message queues (respectively abbreviated IPC MQ and POSIX MQ). Whereas others have been designed for communication over an IP network but are often used for communication on the same host, namely TCP and UDP sockets. The three last mechanisms are parallelism-oriented mechanisms. Barrelfish message passing is a user-level mechanism which has been specifically devised for manycore machines. Open MPI [6] (abbreviated OMPI) and KNEM [3] are two mechanisms that have been designed for multi-processor machines. The former is a user-level mechanism that implements the MPI interface [7], while the latter is a Linux kernel module that is used in conjunction with MPI[1].

Table I summarizes the key characteristics of the studied mechanisms. These characteristics are divided into two categories: those related to the robustness of the different mechanisms (rows 1 to 4) and those related to the performance of the mechanisms (rows 5 to 8).

A. Robustness

We consider that a communication mechanism is robust when faulty user-level processes cannot corrupt messages or data structures used by the communication mechanism, and when user-level crashes of either senders or receivers can not hurt the communication mechanism. Consequently, we report in Table I four different characteristics: the possibility for a faulty user-level process to corrupt messages (row 1), the possibility for a faulty user-level process to corrupt the data structures of the communication mechanism (row 2), the ability to correctly handle crashes of senders processes (row 3), and the ability to correctly handle crashes of receivers processes (row 4). We consider that the crash of a sender process is correctly handled if the receivers associated to that sender are notified of the crash (e.g., with a specific value returned by the receive primitive), and can thus stop waiting for messages from that sender. We say that a receiver crash is correctly handled when senders communicating with this receiver

[1]The choice of evaluating the OMPI implementation of the MPI interface is due to the fact that KNEM supports OMPI.

		Unix domain socket	TCP socket	UDP socket	IPC MQ	POSIX MQ	Pipe	Pipe + `vmsplice()`	Barrelfish MP	OMPI	KNEM	REICoM
1	Faulty user-level processes cannot corrupt messages	√	√	√	√	√	√	-	-	-	-	√
2	Faulty user-level processes cannot corrupt data structures	√	√	√	√	√	√	√	-	-	-	√
3	Crashes of senders are correctly handled	-	√	-	-	-	√	√	-	-	-	√
4	Crashes of receivers are correctly handled	√	√	-	-	-	√	√	-	-	-	√
5	Nb message copies for sending 1 message to N receivers	N	N	N	N	N	N	0	N	N	0	1
6	Nb message copies for receiving 1 message	1	1	1	1	1	1	1	1	1	1	1
7	Nb system calls for sending 1 message to N receivers	N	N	N	N	N	N	2*N	0	0	N	1
8	Nb system calls for receiving 1 message	1	1	1	1	1	1	2	0	0	1	1

Table I: Summary of the key characteristics of state-of-the-art communication mechanisms, and of REICoM.

are notified, and when messages sent to the crashed receiver are garbage collected from the data structures of the communication mechanism.

We can make three observations about the results displayed in Table I. The first observation is that all kernel-level communication mechanisms, but Pipe + `vmsplice()` and KNEM, guaranty that faulty user-level processes can neither corrupt messages, nor data structures used in the mechanism. This comes from the fact that, using these mechanisms, when a message is sent, it is copied from the user-space buffer of the senders process to a kernel-space buffer. This buffer, as well as the data structures of the communication mechanism, are not accessible by user-space processes, and can thus not be corrupted. On the contrary, when using Pipe + `vmsplice()`, KNEM or any of the user-level mechanisms, i.e., Barrelfish MP or OMPI, messages are not copied from the user space to the kernel space. Consequently, senders can accidentally access and modify a message after it has been sent. Further, in Barrelfish MP, OMPI and KNEM, all data structures used in the communication mechanism are kept in the user space and are directly accessible by user-level processes. It is thus possible for faulty processes to corrupt these data structures. This may result in the corruption of messages, in their delivery out-of-order and in message loss.

The second observation we can make is that only TCP, Pipes and Pipes + `vmsplice()` correctly handle the crash of senders processes. The remaining mechanisms do not properly handle the crash of senders. Specifically, Unix Domain sockets, UDP, POSIX MQ, IPC MQ and Barrelfish MP offer two variants of the receive primitive, a blocking and a non-blocking version. In the case of a sender crash, the blocking version results in the receiver to be blocked as the latter will wait for a message forever. The non-blocking version of the primitive only informs the application that there is no message to read. In this situation the application cannot differentiate from a crashed sender and a slow sender. Finally, both OMPI and KNEM kill all the processes that were communicating with a crashed sender. This makes them unusable for many fault tolerance protocols. For instance, in a crash-fault tolerant replication protocol, this would make all the system collapse if only one replica crashes.

The third observation we can make is that the only mechanisms that correctly handle crashes of receivers processes are Unix domain sockets, TCP sockets, Pipes, and Pipes + `vmsplice()`. With the remaining mechanisms, messages are not garbage collected. Consequently, the resource consumption of the system increases as the number of crashed receivers increases. On the long term, this can reduce the amount of memory available for other applications and cause collateral failures (e.g., an application cannot start because it can not create a new communication channel). Regarding OMPI and KNEM, the behavior is similar with the crash of a sender. It results on the killing of all the senders that were communicating with the crashed receiver, which is not acceptable when developing fault tolerance protocols.

B. Performance

We study in this section the performance characteristics of the different mechanisms. In term of performance, we are interested by one-to-many communications, as they are extremely used in fault-tolerance protocols. In the context of inter-core communication, two kinds of operations are particularly costly. In the case of big messages, i.e., more than 1kB, the message

copy is the most costly operation. In the case of small messages, i.e., up to 1kB, performing a system call is the most costly operation. For instance, on the hardware used in Section V, performing a system call is 14 times more costly (in terms of CPU cycles) than invoking a standard function. Table I shows the number of message copies performed for sending one message to N receivers (row 5), the number of message copies performed by one receiver when receiving one message (row 6), the number of system calls performed for sending one message to N receivers (row 7), and the number of system calls performed when receiving one message (row 8).

The first observation we can make is that all the mechanisms but Pipe + `vmsplice()` and KNEM require N message copies for sending one message to N receivers. Furthermore, all mechanisms but Pipe + `vmsplice()`, Barellfish MP and OMPI require N system calls for sending one message to N receivers. As these operations are costly, if one-to-many communications are often used in an application, the latter will suffer unnecessary computational overhead. Some mechanisms avoid this cost for one of the two operations (either system calls or message copies) but they all fail in optimizing them both. For instance, Pipe + `vmsplice()` and KNEM both do not require any message copies for sending a message to N receivers. However, this absence of message copies comes at a cost: for each message, a receiver has to notify the sender that it read the message, thus inducing additional system calls. This explains why, using Pipe + `vmsplice()`, a sender requires 2*N system calls for sending a message to N receivers, and a receiver requires 2 system calls for receiving one message. Similarly, KNEM requires N system calls to send a message to N receivers.

Symmetrically, those mechanisms that optimize the number of system calls required to send a message to N receivers, i.e., user-level mechanisms, fail in optimizing the number of message copies required to send a message. Indeed, both Barrelfish MP and OMPI use N message copies for sending one message to N receivers.

C. Conclusion of the analysis

For reaching satisfactory robustness, a communication mechanism needs to protect sent messages and data structures from faulty user-level mechanisms. This can be done by storing messages and data structures in the kernel level. Furthermore, the mechanism needs to correctly handle the crash of senders (respectively, receivers) by notifying the corresponding receivers (respectively, senders), which allows the application to efficiently handle the crash. From our analysis, only TCP and Pipes offer this level of robustness. All other mechanisms fail to correctly handle sender or receiver crashes. Furthermore, user-level mechanisms fail in protecting the integrity of messages and data structures from faulty user-level processes. Finally, existing mechanisms fail in optimizing both the number of message copies and system calls required to send a message to N receivers, which necessarily impacts performance of protocols that require one-to-many communications, as further demonstrated in Section V.

IV. The REICoM mechanism

REICoM (*Robust and Efficient Inter-core Communication Mechanism*) is a kernel-level communication mechanism dedicated to one-to-many communications between processes residing on the same host. REICoM is robust, as faulty user-level processes cannot corrupt messages or data structures used by the communication mechanism, and as the crash of the sender or the receiver is correctly handled. Moreover, REICoM implements a one-to-many communication primitive that allows sending a message to N receivers using only one system call and one message copy. We start by presenting an overview of REICoM in Section IV-A. We then present its detailed design in Section IV-B. Finally, we present its implementation and optimizations in Section IV-C.

A. Overview

Using REICoM, processes communicate using so-called *communication channels*. Each communication channel is associated to a fixed set of senders and a fixed set of receivers[2]. In order to be efficient, a specific attention in the design has been given to minimize the number of memory copies and of system calls that are performed when sending and receiving messages. For that purpose, a bitmap is associated to each message. This bitmap contains one bit per receiver associated to the communication channel. The bit at position i in the bitmap is set to 1 when the message has been sent and must be read by the i^{th} process of the receiver set. It is reset to 0 when the i^{th} process has read it. Thanks to this bitmap mechanism, only one system call is required to send a message to a set of receivers. Moreover, each sent message is only copied once from the sender user-level buffer to REICoM's data structures.

[2]New communication channels must be created when the number of senders or the number of receivers change.

The key characteristics of REICoM are presented in Table I (in the grey column). Regarding robustness, REICoM guarantees that faulty user-level processes cannot corrupt the messages (row 1), nor the data structures of the mechanism (row 2). This comes from the fact that the communication channels and the mechanism data structures are stored in the kernel memory, which is not accessible by user-level processes. Furthermore, when a process crashes, be it a sender or a receiver, REICoM properly handles this crash (rows 3 and 4) by releasing memory and prevents non-faulty processes to block.

Regarding performance, REICoM is designed in such a way that a message sent to N receivers is copied only once (row 5), and requires only one system call (row 7). Moreover, REICoM requires one message copy for receiving a message (row 6), as well as one system call (row 8).

Summarizing, together with TCP sockets and Pipes, REICoM is the most robust communication mechanism. Nevertheless, REICoM is much more efficient than TCP, Pipes and all the other mechanisms for one-to-many communications. These observations on performance are confirmed by the experimental results presented in SectionV.

B. Detailed design

The main structure of a communication channel is a circular buffer which stores a set of messages. The maximal number of messages that can be stored in the buffer and the maximal size of each message are specified as a parameter of the communication channel creation primitive and cannot be modified after its creation. Each channel uses a variable, named `next_send_entry`, which indicates to the senders the next entry in the circular buffer where to write a message. Multiple senders can concurrently access the channel, thus this variable is protected by a lock[3]. Similarly, each receiver maintains a variable, named `next_read_entry`, which indicates the next entry in the circular buffer where to read a message. These two variables, `next_send_entry` and `next_read_entry`, ensure that the messages are written in and read from the channel in a FIFO order, thus the application does not need to implement a high-level mechanism to reorder the received messages. In order to keep the mechanism simple and efficient, all messages sent on a communication channel are intended to all receivers associated to that channel. Indeed, this avoids costly synchronization operations between senders and receivers to notify the latter that they have messages to read. Consequently, a communication channel must be created for each distinct set of receivers that must receive the same messages.

[3]Note that the number of senders that can concurrently access the channel is limited to the size of the channel, in order to prevent two senders to write their message at the same location.

Figure 1 depicts the steps performed to send and receive a message using REICoM.

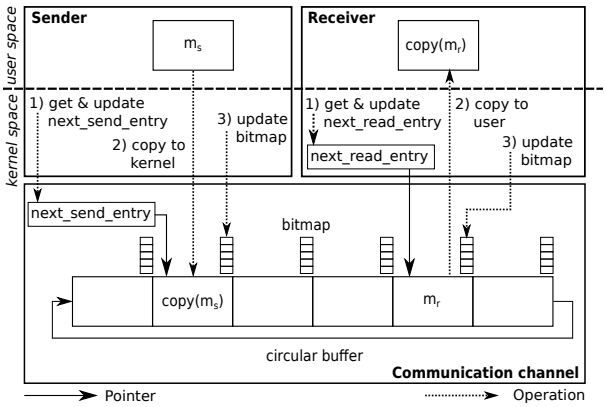


Figure 1: Robust and Efficient Inter-core Communication Mechanism overview.

Sending a message. To send a message, a sender performs a system call, giving as parameters the address and the length of the message. The system call performs the following tasks: it first gets the address of the next available entry in the channel, i.e, `next_send_entry`, and updates this variable by setting it to the next entry in the buffer in order to allow other potential senders to simultaneously use the channel. These two operations are atomically executed. Then, the sender waits until the bitmap associated to the entry it will use only contains null values. When this is the case, it copies the content of the message to be sent from the user-space memory of the sender to the entry. The kernel-space memory is not accessible by user-level processes. As a result, this memory copy into the kernel-space memory of REICoM ensures that the message cannot be corrupted by faulty user-level processes. Then, it updates the bitmap associated with the entry by setting all the bits to 1. Finally, it wakes up the sleeping receivers if there are any.

Receiving a message. To receive a message, a receiver performs a system call that works as follows. It first gets the address of the next available message, which is contained in the `next_read_entry` variable. It then updates this variable. As messages are received in a FIFO order, the next message to be received is the message at the next position in the circular buffer. If the bitmap associated to this entry indicates that there

Function	Operation
`ioctl(`fd`, `$request$`, `$argp$`)`	Depending on the $request$ it creates (CREAT), destroys (DESTR), gets the properties (GET) or modifies (MODIF) a channel (identified by the file descriptor fd). $argp$ is a structure that defines the properties of the channel (max number of senders and receivers, max message size, channel size and channel name). Returns -1 on error, 0 on success.
`open(`$channame$`, `$flags$`)`	Open the channel $channame$ with the access mode defined by $flags$ (i.e., O_RDONLY if opened by a receiver or O_WRONLY if opened by a sender). Returns a file descriptor which identifies the channel.
`close(`fd`)`	Close the channel identified by the file descriptor fd.
`write(`fd`, `buf`, `$count$`)`	Send the message starting at the address buf and of size count to the channel identified by fd. Returns the number of bytes written or -1 on error.
`read(`fd`, `buf`, `$count$`)`	Receive a message from the channel identified by fd and place it in the buffer at address buf of size $count$. Returns the number of bytes read or -1 on error.

Table II: REICoM API.

is a message to receive (i.e., if its associated bit in the bitmap is set to 1), then the receiver copies the message to its private buffer, updates the bitmap, and wakes up the waiting senders if there are any. Otherwise, if there is no message to read, then it waits for a message at this entry.

Handling senders and receivers crashes. Besides storing all data structures and messages in kernel-space in order to prevent faulty processes to corrupt them, REICoM implements specific mechanisms to guarantee robustness in the presence of crash faults. More precisely, when the kernel notifies the crash of a sender or a receiver, REICoM performs a set of actions to guarantee that no process will be blocked, and that allocated memory will be properly released. The actions performed by REICoM are the following ones. If the process that crashed is a sender and if it was the only sender connected to the channel, then all waiting receivers are awaken and the receive operation returns immediately. Consequently, receivers cannot be blocked in the channel, waiting for new messages. If the process that crashed is a receiver, then all the messages that were intended to it are marked as read. This ensures that messages that should have been received by the crashed receiver will be garbage collected as soon as they will have been read by other processes they are intended to.

Handling unresponsive receivers. REICoM may suffer from receivers that stopped reading messages without crashing. To solve this issue, each message is associated with a timer that the sender starts when sending a message. This timer is stopped when the last reader has read the corresponding message. If the timer expires, the application is notified with the set of readers that failed reading before the timeout. The application can use this notification to either increase the value of the timeout or to kill the corresponding readers.

C. Implementation and optimizations

We have implemented REICoM as a Linux kernel module. The benefits of this approach are that it is easy to implement, debug and deploy. Indeed, there is no need to compile the whole kernel and to restart the machine in order to use it.

We have used several data structures and interfaces provided by the Linux kernel. Table II shows the API of REICoM. These methods are traditional methods from the Linux kernel. Note that a process can also use `fcntl()` in order to set the send and receive operations in non-blocking mode, and `select()` in order to wait for a message on multiple channels at once. In addition, we have used the kernel wait queues to make processes wait and to wake them up. More precisely when a process needs to wait, it adds itself to the queue, sets its current state to waiting, and sleeps. Waking up a process is easy: it consists in removing the process from the wait queue and in setting its state to running, so that it can be scheduled again. Moreover, the `next_send_entry` variable is protected by a spinlock, i.e., a lock on which the processes busy-wait, rather than sleep. We chose to use spinlocks because it is more efficient to wait on a spinlock than on a traditional, sleeping lock.

In order to increase the efficiency of REICoM, we have implemented two cache-related optimizations. First, REICoM structures are aligned on a cache line[4]. Second, REICoM structures are padded, i.e., extra bytes are added at the end of the structures, so as the size of each structure becomes a multiple of a cache line size [13]. The advantage of padding is that it prevents false sharing. False sharing occurs when several processes access different unrelated data that fit in the same cache line. Although they do not access the same data, the cache coherency mechanism will invalid the entire line each time a single bit is modified, which decreases performance.

[4]Barrelfish MP also uses this optimization.

V. Performance comparison

In this section, we present a performance analysis of REICoM. We compare its performance to that achieved by the state-of-the-art mechanisms presented in Section III. We start by a description of the hardware and software settings we used. Then, we evaluate the mechanisms by implementing a consensus and a snapshot protocol.

A. Hardware and software settings

We run our experiments on an HP Proliant DL165 G7 machine that has two AMD Opteron 6164HE processors clocked at 1.7GHz and 48GB of RAM. Each processor contains two sets of six cores that share a L3 cache of 6MB. Moreover, each core has private L1 and L2 caches of 64kB and 512kB, respectively. We use a Linux kernel version 3.0.0 and profiled the mechanisms with Oprofile [12] and LIKWID [15]. We have evaluated the Open MPI v1.5.4 library, to which we refer by OMPI, and the version 0.9.8 of KNEM. We configure all the communication mechanisms in order to obtain the best performance for each of them. Finally, we have implemented Barrelfish MP in Linux in user-space from the Barrelfish OS source code[5].

B. Consensus protocol: PaxosInside

Guerraoui and Yabandeh have recently proposed PaxosInside [8], an adaptation of the Paxos protocol [11] for manycore machines. Similarly to Paxos, the PaxosInside protocol distinguishes three roles for nodes taking part in the protocol: proposer, acceptor and learner. An important difference with Paxos is that PaxosInside relies on a single active acceptor, which is replaced by a backup acceptor when a failure occurs. For every consensus, PaxosInside performs several rounds of one-to-one and one-to-many communications. More precisely, assuming that there are l learners in the system, every consensus requires $2+l$ one-to-one message exchanges and 1 one-to-many message exchange.

In the presented experiments, we implement a version of PaxosInside per communication mechanism (the difference between these versions being in how they send and receive messages) and deploy it with one proposer, one acceptor, and three learners (which allows tolerating the failure of exactly one learner). The proposer issues requests for consensus in an open loop, i.e., the proposer does not wait for a response before proposing a new value. We wait for 100,000 accepted proposals and measure the peak throughput at which consensus are performed as a function of the request size (from 64B to 1MB). The results are the average over three runs, for which the standard deviation is less than 1% and is thus not depicted.

Figures 2 and 3 present, respectively, the throughput of all state-of-the-art communication mechanisms for small and big message size, in addition to the corresponding throughput improvement of REICoM over these mechanisms. As shown in both figures, REICoM provides a better throughput than the state-of-the-art communication mechanisms for all message sizes. In particular, we observe in Figure 3 that compared to the most robust protocols, i.e., TCP and Pipes, REICoM throughput improvement ranges between 69% to 920% for TCP and between 29% and 190% for Pipes. The minimal throughput improvement reached by REICoM compared to all protocols and all message sizes is 9%, when compared to Pipe + `vmsplice()` for messages of 1MB. Moreover, we observe that the maximal throughput improvement reached by REICoM compared to all protocols and all message sizes is 1389% when compared with KNEM.

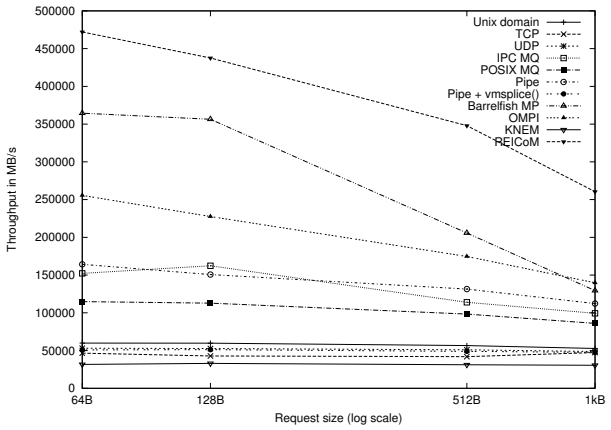
The two design points responsible for the better performance are as expected the small number of message copies and system call performed by REICoM. To assess this, we ran an experiment where the message was copied N times in the channel and found that the throughput of the protocol decreased by 21% at 10kB. Similarly, we performed another experiment with messages of 64B, where the sender was calling the send primitive N times to send a message. We observed that the throughput has dropped by 46%.

More generally, we observe in Figure 2 that the throughout of the protocol using all the mechanisms decreases when the message size increases. This is due to the fact that contrary to small messages, large messages can not be kept for long enough in the cache, which forces readers to fetch them in memory. For instance, we monitored the number of cache misses at the L1 cache when running the two experiments with messages of 10kB and 100kB. We observed that the cache miss rate increases by 439% in the experiment with messages of 100kB compared to the experiment with messages of 10kB.

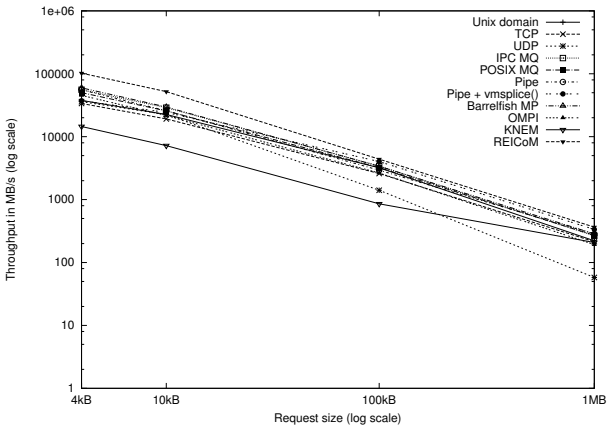
C. Snapshot protocol

The second protocol we use to evaluate REICoM is a snapshot protocol. The second protocol we use to evaluate REICoM is the snapshot protocol proposed by Manivannan and Singhal [14]. Amongst the existing snapshot protocols (e.g., [10]), this protocol is interesting due to the fact that it is lightweight: processes do not need to be synchronized and the dependency between

[5]http://www.barrelfish.org



(a) Throughput of the state-of-the-art communication mechanisms and REICoM for small messages (up to 1kB).



(b) Throughput improvement of REICoM over state-of-the-art communication mechanisms for big messages (from 4kB up to 1MB)

Figure 2: Agreement protocol.

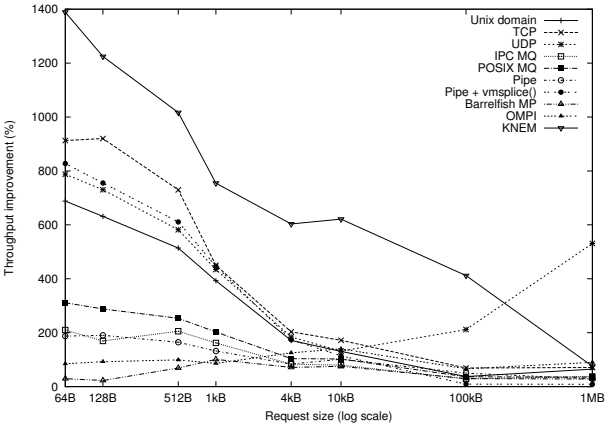


Figure 3: Agreement protocol: Throughput improvement of REICoM over state-of-the-art communication mechanisms

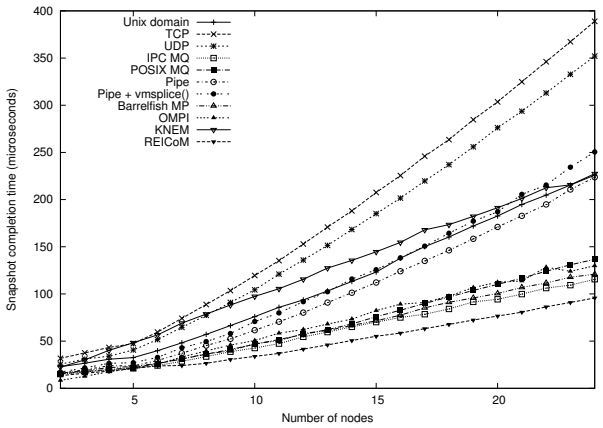
the exchanged messages does not need to be tracked, e.g., using vector clocks. Moreover, it is very efficient because all the messages can be piggy-backed to the messages exchanged by the checkpointed application. It works as follows: to gather a snapshot, a node sends a message to every other nodes in the system. Upon reception of such a message, the nodes send back their latest checkpoint to the initiator of the snapshot. The snapshot is finished once the initiator has received a checkpoint for every node in the system. This protocol involves 1 one-to-many and $n-1$ one-to-one exchanges of messages[6], where n is the number of nodes taking part in the protocol.

In the presented experiments, there is one node that issues 100,000 requests for snapshots to $n-1$ other nodes in a closed-loop, meaning that it only issues a new snapshot request when the previous one has completed. We vary the number of nodes, from 2 to 24. The size of a snapshot request is always 128B and the size of the checkpoints is fixed to 4kB. This size is representative of the quantity of information checkpointed by traditional applications. We measure the time required to complete a snapshot, i.e., the time elapsed between the sending of the snapshot request and the reception of the last checkpoint. Results are all the average over three runs. The standard deviation, of less than 1%, is not shown.

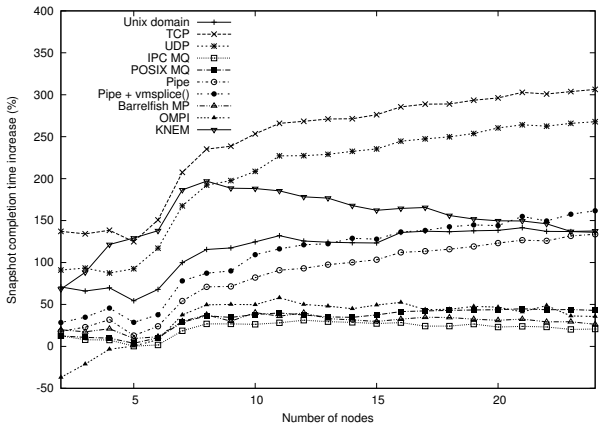
Figures 4a and 4b present, respectively, the snapshot completion time and the snapshot completion time increase of the state-of-the-art mechanisms over REICoM. We observe in both figures that for small number of nodes, i.e., from 1 to 5, only OMPI exhibits better performance than REICoM. Furthermore with this number of nodes, REICoM slightly outperforms IPC MQ, Posix MQ, Barrelfish MP, Pipes, and Pipes + `vmsplice()` (less than 50% difference) and largely outperforms the remaining mechanisms (from 50% to 140% difference). From 5 nodes onward, REICoM consistently outperforms all the existing mechanisms. In particular, compared to TCP and Pipes, which are the mechanisms that provide the same level of robustness as REICoM, we observe that the snapshot completion time improvement of REICoM ranges between 120% and 300% for TCP and 9% and 140% for Pipes.

More generally, we observe from Figure 4a that

[6]The communication from the initiator node to himself is handled internally by the application.



(a) Snapshot completion time of the state-of-the-art mechanisms and REICoM.



(b) Snapshot completion time increase of the state-of-the-art mechanisms compared to REICoM.

Figure 4: Snapshot protocol.

the completion time of all protocols increases with the increase of the number of nodes involved in the checkpointing protocol. This is due to the fact that the snapshot initiator needs to receive more messages in order to complete a snapshot. Since it is a mono-threaded process, it reads sequentially the messages from the different nodes, which takes a longer time.

Finally, we observe that REICoM scales better than the other mechanisms. Indeed, between 2 and 24 nodes, the latency increases by x7, while it increases by x7.5 for Barrelfish MP and up to x15 for OMPI.

VI. Related work

In 2003, Immich *et al.* have presented a study of five inter-process communication mechanisms [9]. More precisely, they have evaluated pipes, named pipes, IPC message queue, shared memory with semaphores and Unix domain sockets on four versions of Linux and two of FreeBSD. They considered a micro-benchmark, with a producer/consumer scheme, for messages ranging from 64B to 4608B. The main focus of this work was to study the impact of the operating system on the performance of existing inter-process communication mechanisms. Wright *et al.* [16] conducted a performance analysis of pipes, Unix Domain sockets and TCP sockets on different manycore platforms. The authors conducted their experiments on different hardware, but focused on micro-benchmarks with only one sender and one receiver and large messages (ranging from 1MB to 100MB). Our performance analysis of existing inter-process communication mechanisms is complementary to the above mentioned studies. None of the above works analyzes the robustness of existing inter-process communication mechanisms and how they behave on manycore machines when they are used for one-to-many communications, which has been the focus of our study. The experimental operating system Singularity OS provides an efficient, zero-copy, communication mechanism [5]. To achieve this goal, it relies heavily on the compiler and the runtime. In particular, static verifications ensure that a process does not access the memory area of a message after it has been sent. The communication mechanism used in Singularity OS is robust. Indeed, the messages and the data structures cannot be corrupted, the crashes of the processes are correctly handled by the mechanism, and it provides a FIFO, loss-less channel between two endpoints. However, this comes at a non-negligible cost: the operating system and the applications have to be developed in a special language, namely Sing#. Consequently, this mechanism cannot be used by legacy applications running on Linux platforms. In contrast, REICoM provides a standard interface (leveraging well-known system calls) and can thus be used in existing systems without modifications.

VII. Conclusion

In this paper, we have studied ten existing communication mechanisms for communications between processes residing on the same machines. Our study has shown that existing mechanisms are either not robust, or not efficient. We have thus proposed REICoM, a new communication mechanism. This mechanism is robust: it tolerates the crash of both receivers and senders, and protects against faulty processes trying that could corrupt data structures or messages. Moreover,

REICoM is efficient. In particular, it provides a one-to-many communication primitive requiring only one message copy and one system call. We have evaluated the different communication mechanisms using two distributed algorithms: PaxosInside — a consensus algorithm for manycore machines —, and a snapshot algorithm. These two applications use a minority of one-to-many communications (five one-to-one and one one-to-many communication for PaxosInside, one one-to-many communications and up to twenty-three one-to-one communications for the snapshot protocol). In both these applications using an optimized one-to-many communication mechanism makes a tremendous difference in the performance of the whole protocol. Indeed, if we specifically look at the two mechanisms that are as robust as REICoM, we conclude that REICoM outperforms them in the two benchmarks by up to 920% and 300% for TCP and 190% and 140% for Pipes in the two benchmarks, respectively.

ACKNOWLEDGMENTS

We would like to thank Fabien Gaud, Baptiste Lepers, Alessio Pace, Nicolas Palix and Sylvain Genevès for their helpful feedback on this work.

The experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

REFERENCES

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1:11–33, January 2004.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009.

[3] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, 2009.

[4] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008.

[5] Manuell Fhndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of Eurosys 2006*. ACM, 2006.

[6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[7] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[8] Rachid Guerraoui and Maysam Yabandeh. PaxosInside. Technical Report EPFL-REPORT-153309, EPFL, 2010.

[9] Patricia K. Immich, Ravi S. Bhagavatula, and Ravi Pendse. Performance analysis of five interprocess communication mechanisms across unix operating systems. *J. Syst. Softw.*, 68:27–43, 2003.

[10] Qiangfeng Jiang, Yi Luo, and D. Manivannan. An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. *Journal of Parallel and Distributed Computing*, 68(12):1575 – 1589, 2008.

[11] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[12] J. Levon. OProfile.

[13] Tongping Liu and Emery D. Berger. Sheriff: Detecting and eliminating false sharing. Technical Report UM-CS-2010-047, University of Massachusetts, Amherst, 2010.

[14] D. Manivannan and M. Singhal. Asynchronous recovery without using vector timestamps. *J. Parallel Distrib. Comput.*, 62:1695–1728, 2002.

[15] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, 2010.

[16] Kwame-Lante Wright and Kartik Gopalan. Performance analysis of inter-process communication mechanisms. Technical Report TR-20070820, Binghamton University, 2007.