Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016
© 2016 The authors and Open Channel Publishing Ltd. All rights reserved.

Broadcasting in CSP-Style Programming

Brian VINTER [1], Kenneth SKOVHEDE, and Mads Ohm LARSEN
University of Copenhagen, Niels Bohr Institute

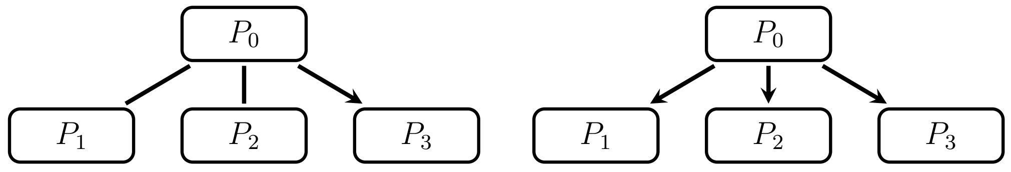
Abstract. While CSP-only models process-to-process rendezvous-style message passing, all newer CSP-type programming libraries offer more powerful mechanisms, such as buffered channels, and multiple receivers, and even multiple senders, on a single channel. This work investigates the possible variations of a one-to-all, broadcasting, channel. We discuss the different semantic meanings of broadcasting and show three different possible solutions for adding broadcasting to CSP-style programming.

Keywords. CSP, JCSP, broadcasting

Introduction

The concept of broadcasting — emitting a message from one process and receiving in all other processes in a group — has been debated in [1] and [2]. The work on Synchronous Message Exchange, SME [3] stems from a lack of broadcasting in CSP. In this work the authors will seek to establish the meaning of broadcasting and the possible benefit of a broadcast mechanism in CSP-style programming.

Perhaps the easiest way to introduce the meaning of a broadcast mechanism is to compare it with the well established one-to-any or any-to-any mechanism which all modern CSP-style libraries offer [4,5,6,7]. With the any-to-any channel a message sent by one process is delivered to any process that is ready to receive. Neither one-to-any nor any-to-any channels are part of the CSP-theory, however they can be emulated using multiple channels [8]. Contrarily a broadcast would be a one-to-all/any-to-all operation where every receiving process on the channel would get a copy of the message. Figure 1a and 1b seek to sketch the difference between a to-any and a to-all send operation.



(a) With the one-to-any model a message ends up at one process in the group

(b) The one-to-all delivers the message to all members of the group.

Figure 1. One-to-any and one-to-all communication.

Broadcasting is a common feature in message based parallel programming libraries, such as MPI [9] and PVM [10]. The need for broadcasting stems from algorithms that

[1]Corresponding Author: *Brian Vinter, Blegdamsvej 17, 2100 Copenhagen OE* E-mail: `vinter@nbi.ku.dk`.

need global synchronization. Broadcast may be used directly, that is for distributing a new global bound value in a parallel branch-and-bound algorithm, or it may be used in combination with a reduction, that is for determining the global change in a system after an iteration in a converging algorithm.

In SME, broadcasts were needed for a set of processes to all know a given value for the next simulated time-step, that is as a stand-alone broadcast. The use of reductions followed by a broadcast is mostly known in performance oriented parallel programs, though applications in the concurrency domain should not be entirely ignored. This work however, only investigates the feasibility of a pure broadcasting mechanism in a CSP-style library.

1. Broadcasting

Broadcasting as a concept appears deceptively simple however, several variations exists that have a slightly different semantics. Fundamentally broadcasting is based on physical broadcasts, that is a sender transmits a message for anybody to receive, as shown in Figure 2.

Figure 2. Physical broadcast mechanism.

We may define *simple broadcast* as the basic mechanism shown in Figure 2; one process may broadcast, any other process may receive. Thus, the *simple broadcast* mechanism has no delivery guarantee, a broadcast that is received by no process is still defined as a correct broadcast. Such a broadcast mechanism is of little use in real-world scenarios, and is thus often not provided. UDP/IP datagrams may be broadcast and if so it is done as *simple broadcast*, that is any layer in the network stack may choose to not propagate the broadcast. The only guarantee that the *simple broadcast* provides is message integrity, that is the message is either delivered in full and as originally sent, or not at all.

Simple broadcast may be improved to be a *reliable broadcast.* A reliable broadcast mechanism has the added semantics that when a message is broadcasted, all processes must receive a correct version of that message. Reliable broadcast does not guarantee any ordering, that is two concurrent broadcasts by two different processes may be received in different order by different processes in the system. While reliable broadcasting may be physically intuitive, as sketched in Figure 3, the lack of total ordering is still a limitation that makes programming harder in various cases.

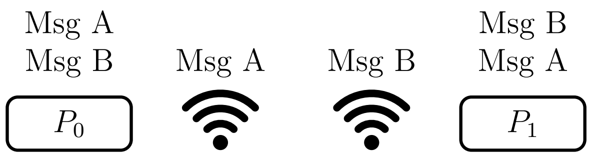


Figure 3. Two senders transmits messages A and B at the same time, because of physical proximity process zero will receive the messages in the order A then B, while process one will receive them in the order B then A. This is still a correct reliable broadcast.

An example where reliable broadcast is not sufficient is described as follows: a system consists of a set of fire detectors, fire alarms, and control boards. Imagine that a detector, A, detects a false fire, that is a forgotten toaster, the person using the toaster immediately cancels the alarm using the control board A', but then another fire detector, B, detects a real alarm. If the broadcast message from B is received before the cancelation from A' then a non-counting alarm would falsely turn off, even though a fire was indeed spreading.

Reliable broadcasting may be further improved upon to guarantee total ordering; this type of broadcast is known as *atomic broadcast*. In the *atomic broadcast* all broadcasts are received by all processes, and in the same order. This also implies that a process that has failed, that is been unable to receive for some reason, cannot recover, but must leave the system and, if possible, rejoin. If a system implements *atomic broadcasts* the two processes in Figure 3 will agree on which order the messages A and B are received. Whether they are received as A then B or as B then A is still not defined, only that all processes will receive them in the same order.

A fourth broadcast mechanism which is well researched is the *causal broadcast* [11]. In this model a broadcast message B cannot be delivered to a receiver ahead of another message, A, if message A was received by the broadcaster of message B prior to that broadcast. Causal broadcasts however, has turned out to be very complex to implement and harder to work with than *atomic broadcasts*, thus we will not investigate *causal broadcasts* in this work.

Apart from delivery guarantees and message order broadcasts may be defined as synchronous or not. A *synchronous broadcast* will only be delivered to a receiver once all receivers are ready to receive, while an asynchronous delivery simply guarantees either reliable or *atomic broadcasts*. In distributed systems *synchronous broadcasts* are not available in any widespread libraries as the message cost becomes prohibitive, but in a CSP-library context a *synchronous broadcast* could be more efficiently implemented.

Finally, conventional broadcast literature differentiates between broadcasts in open and closed groups [12]. Open group broadcast means that a process, which is not on the list of recipients may still broadcast a message to the group, while closed group broadcasts require the sender to be a member of the recipient group.

2. Broadcasting in Message Passing Systems

2.1. Parallel Virtual Machines

The Parallel Virtual Machines library, PVM, supports group communication from version 3. Since PVM has a rather low level approach to message passing messages must first be packed into a message buffer and can then be broadcast to a group. The below example in listing 1, adapted from the PVM `man` page, packs 10 integers from a variable called array and then broadcasts the values, using the tag 42 to a group of processes, called tasks in PVM, named worker.

```
info = pvm_initsend(PvmDataRaw);
info = pvm_pkint(array, 10, 1);
info = pvm_bcast("worker", 42);
```

Listing 1. PVM broadcast sender.

The receiver will then have to issue an ordinary receive and then unpack the data as shown below in listing 2.

```
buf_id = pvm_recv(&tid, &tag)
info   = pvm_upkint(array, 10, 1)
```

Listing 2. PVM broadcast receiver.

In PVM the messages are received by the individual recipients as ordinary messages. PVM broadcasts are open group and provides only reliable broadcasts. In addition to the broadcast operation PVM also provides a multicast operation where a group is dynamically created from a list of recipients.

2.2. Message Passing Interface

Message Passing Interface, MPI, manages broadcasts rather differently than PVM. Groups are defined as subgroups of the overall set of processes, called `MPI_COMM_WORLD`. Message contents is defined like ordinary point-to-point messages, and addressing is simply to a subgroup. The major difference from PVM is that the broadcast is issued by all participants in the group, and a parameter in the broadcast specifies which process is the sending and all others are then receivers. This approach means that broadcasting in MPI is in closed groups only. A ten integer dense array broadcast example, from process zero to all other processes in the program, then looks as below in listing 3.

```
result = MPI_Bcast(data, 10, MPI_Int, 0, MPI_COMM_WORLD)
```

Listing 3. MPI broadcast.

3. Models for Broadcasting in CSP

From the existing systems that features a broadcast method, it is clear that such a mechanism has a use and this would be interesting to provide in a CSP-style library as well. Merging broadcast with CSP-semantics is not trivial, however, as we have previously shown in the SME work [3]. In this section we sketch the various broadcast styles one may imagine for CSP, and try to evaluate their feasibility for CSP.

3.1. Broadcast Messages

The simplest approach would be to introduce a broadcast message command, `channel.bcast(msg)`, similar to the semantics in PVM. With this approach, the library has to know about all processes that holds a reading end of the channel and then relay the message to each such process as they issue read commands from the channel. This approach raises a set of difficulties however; CSP dictates that the broadcast operation does not return until after the operation has finished, that is when all processes has received the message that was broadcast. However, it is not intuitive what should happen to the channel while the broadcast completes, if we require the channel to be blocked until the message is received everywhere then deadlocks may arise as other processes may have agreed to exchange a point-to-point message on the channel. If, on the other hand, we do not freeze the channel while the broadcast completes, other patterns that are non-intuitive may occur. If one process, that holds a receiving end of a channel, never issues a read on that channel then the channel will require infinite buffer-capacity to remain functional. A broadcast may be received by all but one process, and other messages have been passed as point-to-point afterwards, but if the last process then terminates, the broadcast has never truly been completed and the meaning of a broadcast becomes weaker.

An emulation of a broadcast message could be implemented as in Figure 4. This naïve network, and the theory behind, will be discussed in the next sections.

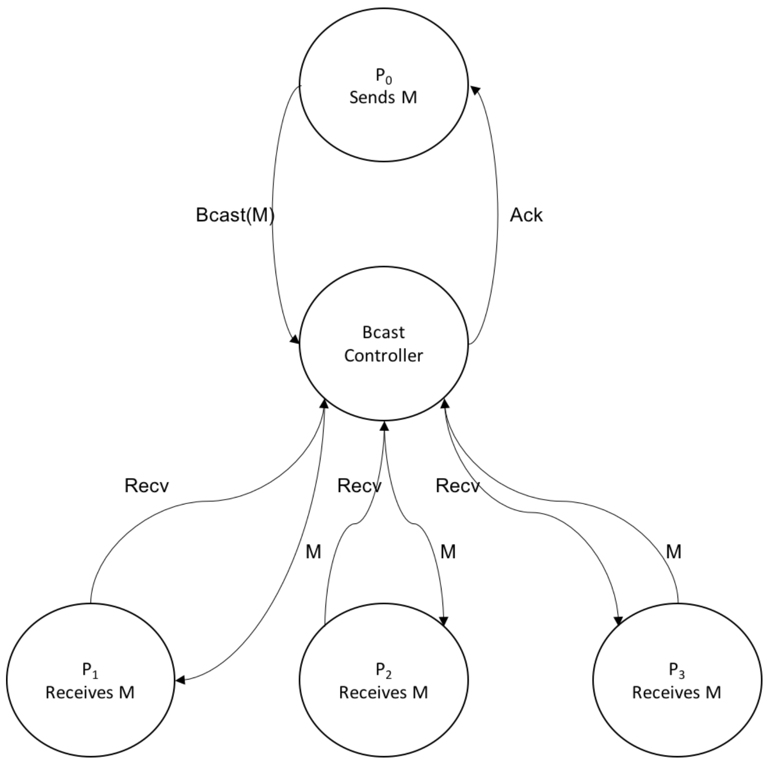


Figure 4. A naïve network for emulating synchronous broadcasting in CSP.

3.2. Mailboxes

A slightly more complex approach would be to introduce a mailbox system, similar to mailboxes as they are found in Erlang, but with a subscription service so that a set of processes can read the same message. A mailbox model is very intuitive for a programmer to comprehend; a message is sent to a central position and all processes may read it. However, this model does not include any synchronization as a CSP-style programmer would expect. If the model is extended to synchronize — either by announcing to the sender when all processes have read the message, or by introducing a fully synchronous model — then the mailbox model is identical to the broadcast message as described above.

A network using mailboxes for broadcast is shown in section 3.4.

3.3. Broadcast channels

A third approach to CSP-style broadcasting is to introduce dedicated broadcast channels. Such a broadcast channel has semantics that are identical to the synchronous broadcast message in the first scenario. The difference is that with a broadcasting channel there are no point-to-point messages that may be interleaved with broadcasts. This way the deadlock scenario from the broadcast message cannot occur, the channel provides synchronized communication as a point-to-point channel and may be used in guarded expressions just as an ordinary channel.

The downside of this approach is that programmers must use another type of channel. In JCSP this inconvenience is small as there is by design a large number of channel types, in PyCSP the change is more radical as the single channel type must be abandoned.

3.4. Broadcast in CSP-theory

In the following section, primed processes (i.e. S') will be used to show that the process can live on afterwards.

Emulating broadcast channel in CSP-theory can be done with n processes. Here S can broadcast a message to all of $P_{0..n}$:

$$S = m!x \rightarrow S'$$
$$P_i = m?x \rightarrow P'_i$$

$$S \parallel \left(\mathop{\parallel}_{i=0}^{n} P_i \right)$$

However, this is possible only, if we disregard the fact, that only two processes are allowed to communicate with each other at a given time according to Hoare CSP [13].

If we instead want to broadcast, but adhere to the original theory, we have to do something else. We will create a network of processes, where S will act as sender, B_c will act as a broadcast controller and $P_{0..n}$ will be receivers. They will pass messages using a two-phase commit protocol.

$$S = m!x \rightarrow m_{\text{ACK}} \rightarrow S'$$
$$B_c = m?x \rightarrow \left(\mathop{|||}_{i=0}^{n} c_i!x \rightarrow c_{i,\text{ACK}} \rightarrow \checkmark \right) ;\ m_{\text{ACK}} \rightarrow B'_c$$
$$P_i = c_i?x \rightarrow c_{i,\text{ACK}} \rightarrow P'_i$$

$$S \parallel B_c \parallel \left(\mathop{\parallel}_{i=0}^{n} P_i \right)$$

Looking at the system with FDR [14] in Listing 4 we see that it is deadlock free. A trace were verified where the message was received out-of-order.

```
N = 3
PNAMES = {0..N-1}
MSG = {"MSG"}

channel mack
channel m:MSG
channel cack:PNAMES
channel c:PNAMES.MSG

S(x) = m!x -> mack -> S(x)
B = m?x -> (||| i:PNAMES @ c.i!x -> cack.i -> SKIP) ; mack -> B
P(i) = c.i?x -> cack.i -> P(i)

SYSTEM(x) = S(x) [|{|m, mack|}|]
                 (B [|{|c, cack|}|] || i:PNAMES @[{|cack, c.i|}] P(i))

assert SYSTEM("MSG") :[deadlock free [F]]
assert SYSTEM("MSG") \ {|m, mack, cack|}
  :[has trace [T]]: <c.0."MSG", c.1."MSG", c.2."MSG">
```

Listing 4. FDR3 verified Broadcasting CSP-system.

This of course means that the broadcast controller must know how many receivers there are present in the system and be able to pass messages to all of them on their respective channels.

If we want to allow all processes to be the writer at a given time, we can model this as alternation on the communication:

$$P_i = \Big(c_i!x \to c_{i,\text{ACK}} \to P'_i\Big) \Box \Big(c_i?x \to c_{i,\text{ACK}} \to P''_i(x)\Big)$$

$$B_c = \mathop{\Box}_{i=0}^{n} \left(c_i?x \to \left(\mathop{\|}_{\substack{j=0\\ j\neq i}}^{n} c_j!x \to c_{j,\text{ACK}} \to \checkmark\right) ; c_{i,\text{ACK}} \to B'_c\right)$$

$$B_c \parallel \left(\mathop{\|}_{i=0}^{n} P_i\right)$$

Here all the processes either write on their channel or read from their shared channel with B_c.

The equivalent mailboxing scenario can be made with a mailbox process for each receiving process. Here the writer will be able to continue, once all mailboxes have ACK'ed back that they have received the message:

$$S = b!x \to b_{\text{ACK}} \to S$$

$$B = b?x \to \left(\mathop{|||}_{i=0}^{n} m_i!x \to m_{i,\text{ACK}} \to \checkmark\right) ; b_{\text{ACK}} \to B$$

$$M_i(x : xs) = \Big(m_i?y \to m_{i,\text{ACK}} \to M_i(x : xs : y)\Big) \Box \Big(c_i!x \to c_{i,\text{ACK}} \to M_i(xs)\Big)$$

$$P_i = c_i?x \to c_{i,\text{ACK}} \to P_i$$

$$S \parallel B \parallel \left(\mathop{\|}_{i=0}^{n} M_i(\emptyset) \parallel P_i\right)$$

where : means CONS and $x : xs$ is the head and tail of the message list. If one does not need the guarantee on each mailbox having received the message, the ACK steps can be omitted. This has been tested with FDR and found to be deadlock free. A trace was also found with each process having received the message.

The mailboxing works for arbitrarily big buffers; in Listing 5 it is shown with a buffer size of 5.

```
N = 3
MAXBUFFER = 5
PNAMES = {0..N-1}
MSG = {"MSG"}
```

```
channel back
channel b:MSG
channel cack, mack:PNAMES
channel c, m:PNAMES.MSG

S(x) = b!x -> back -> S(x)
B = b?x -> (||| i:PNAMES @ m.i!x -> mack.i -> SKIP) ; back -> B

M(i, <>)      = m.i?y -> mack.i -> M(i, <y>)
M(i, xss)     = if #xss > MAXBUFFER then M1(i, xss) else Ms(i, xss)
M1(i, <x>^xs) = c.i!x -> cack.i -> M(i, xs)
Ms(i, xss)    = M1(i, xss) [] (m.i?y -> mack.i -> M(i, xss^<y>))

P(i) = c.i?x -> cack.i -> P(i)

MAILBOX = ||| i:PNAMES @ M(i, <>)
RECV    = ||| i:PNAMES @ P(i)
COMM(x) = S(x) [|{|b, back|}|] B

SYSTEM(x) = (COMM(x) [|{|m, mack|}|] MAILBOX) [|{|c, cack|}|] RECV

assert SYSTEM("MSG") :[deadlock free [F]]
assert SYSTEM("MSG") \ {|b, back, m, mack, cack|}
  :[has trace [T]]: <c.1."MSG", c.0."MSG", c.2."MSG">
```

Listing 5. FDR3 verified Mailboxing CSP-system.

4. Related work

Both JCSP [1] (Java Communicating Sequential Processes) and CHP [2] (Communicating Haskell Processes) offers a form of broadcast channel. In the former, a "one-to-many" channel is implemented. This must know the number of readers when initialized, and works by having two barriers, one before read and one after, so that all readers are done reading, before the writer is released. The latter is implemented in a similar way, where each reader enrolls to receive the same value from a single writer.

5. Conclusion

Broadcasting in CSP-style has been frequently discussed, and in the SME work replaced by a bus-style channel [15]. While adding broadcasting to CSP-style programming appears appealing and straight forward it has yet not been added to any CSP-style library. In this work we have outlined three approaches to adding broadcasting to CSP, and concluded that while they are all possible, only the explicit broadcasting channel is able to provide what the authors consider a CSP-style behavior and the common expected functionality of a broadcast operation. The need for broadcasting in CSP-style libraries is still an issue that must be investigated further, it is not obvious that broadcasting has a general use in application where CSP is commonly used, but might be reserved for fore traditional HPC style applications.

References

[1] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting barriers: synchronisation with choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22(8):1049–1062, 2010.

[2] Neil Christopher Charles Brown. *Communicating Haskell Processes*. Citeseer, 2011.

[3] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2015*, pages 201–212, aug 2014.

[4] Nan C. Schaller, Gerald H. Hilderink, and Peter H. Welch. Using Java for Parallel Computing - JCSP versus CTJ. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226, Sep 2000.

[5] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 263–276, Nov 2009.

[6] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, jul 2007.

[7] Kenneth Skovhede and Brian Vinter. CoCoL: Concurrent Communications Library. In *Communicating Process Architectures*, 2015.

[8] Mads Ohm Larsen and Brian Vinter. Exception Handling and Checkpointing in CSP. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 201–212, aug 2012.

[9] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[10] Al Geist. Pvm (parallel virtual machine). In *Encyclopedia of Parallel Computing*, pages 1647–1651. Springer, 2011.

[11] Robbert van Renesse. Why bother with catocs? *ACM SIGOPS Operating Systems Review*, 28(1):22–27, 1994.

[12] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[13] Charles Antony Richard Hoare et al. *Communicating Sequential Processes*, volume 178. Prentice-hall Englewood Cliffs, 1985.

[14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika brahm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[15] Brian Vinter and Kenneth Skovhede. Bus centric synchronous message exchange for hardware designs. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2015*, pages 245–257, aug 2015.