Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

Extensions to the Concurrent Communications Library

Kenneth SKOVHEDE [1] and Brian VINTER

Niels Bohr Institute, University of Copenhagen

Abstract. This paper presents updates and measurements for the Concurrent Communications Library, CoCoL, which is a CSP inspired library targeting C# and other languages running on the Common Language Runtime, also known as .Net. We describe the new library interface methods that simplify writing correct, encapsulated and compositional networks. We also describe an extension to the library, which enables communication over network connections and measure the performance.

Keywords. CSP, concurrent programming, process oriented programming, C#, .Net, Common Intermediate Language, distributed systems, network communication

Introduction

The Concurrent Communications Library, CoCoL, was introduced in 2015 [1] and is a CSP [2] inspired library targeting C# and other languages running on the Common Language runtime, also known as CLR, .Net or CIL. The driving idea in CoCoL is to provide a simple interface for constructing programs using CSP techniques, but without requiring knowledge about the history and theory of CSP. Hopefully this approach makes it more accessible to programmers, which are unfamiliar with CSP, as they only need to understand a *channel* as an abstraction. The API in CoCoL is primarily a class that is aptly named `Channel`, which has equally well-named methods called `Read` and `Write`. The methods `ReadFromAny` and `WriteToAny` provide support for CSP-style *alternation* without explicitly exposing a CSP `Guard`. Unlike other CSP libraries, there is no special `Process` class, instead, any code is allowed to interact with a channel and can be interpreted as a kind of implicit process.

As noted with other library implementations, such as JCSP [3] and CPPCSP [4], representing a CSP process as a thread comes with a significant memory footprint, limiting the number of processes to less than 100.000, even on modern hardware. In CPPCSP, this can be optimized by setting the thread stack, and by manually setting processes to share a single thread. The CoCoL solution to this issue is to utilize the `await` and `async` keywords in the C# language to return `Task` objects representing a future result. The CLR implementation of `await` allows multiple operations to wait on different future results, but does not tie up the calling thread, thus transforming the program to something similar to a *bag-of-tasks* execution model. In this model, a thread-pool will pick operations that can be completed and execute them with available threads, potentially starting more threads if required. This combination of a thread-pool, `await`, and `Task` results in a program that looks sequential, but actually runs in parallel, while maintaining a high level of parallelism and a low overhead for each channel communication. Like in ProcessJ [5], this decoupling from threads allows millions of concurrent processes to run on modest hardware.

[1]Corresponding Author: *Kenneth Skovhede, Niels Bohr Institute, Blegdamsvej 17, DK-2100 Copenhagen OE.* Tel.: +45 35325209; E-mail: `skovhede@nbi.ku.dk`.

Motivated by the promising performance results from the initial implementation, we have updated CoCoL with a number of features that aim at making CoCoL more feature rich and productive, which are described in section 1. In section 2 we describe a channel implementation that is a drop-in replacement for the normal channel, but sends requests over a `TCP` connection, such that the actual channel can be serving requests from multiple machines. The implementation of a *network channel* utilizes the *two-phase-commit* [6] strategy already found in CoCoL, allowing network channels to participate in *alternation* constructs on the same terms as local channels.

1. New Library Constructs

After we introduced CoCoL, we implemented various improvements and extensions to the core library, to make it simple to work with, without sacrificing the predictability. The changes are almost exclusively implemented as extra functions, leaving the core channel logic and implementation unchanged from the initial description [1].

1.1. Mixed Read and Write Requests

In the initial version, CoCoL supported alternation requests by providing a list of channels and an external choice strategy to either `ReadFromAny` or `WriteToAny`. These methods run through the list of channels in the order given by the external choice strategy, and registers a request to either read or write, and guarantees that at most one read or write is performed before an optional timeout occurs. To ensure that only a single read or write is performed, each request is supplied with a shared instance of a `SingleOffer` class which provides a *Two-Phase Commit* entry. As the CoCoL `Channel` uses the *Two-Phase Commit* entry to query a request when pairing reads and writes, this is handled with a single lock inside the `SingleOffer` instance, and thus does not require cooperation between participating channels.

Using this same technique, we have added the `ReadOrWriteAny` method, which allows mixing read and write requests. Unlike `WriteFromAny`, we have extended the method to allow writing *different* values to each of the channels. Consider the simple network in listing 1, which provides the current and previous value to potential readers, and allows a writer to set the current value. As shown in the example, each channel in the list must be marked, such that it is possible to determine if the desired operation is a read or a write, and for the write operation also what value to write, should the operation be performed. The implementation of this method is almost identical to the `ReadFromAny` and `WriteToAny`, except that a `MultisetRequest` instance is passed in place of the channel, such that the appropriate method can be invoked on the channel, and, for a write operation, send the specified value.

A related operation on multiple channels would be to perform operations with different types for each operation. One simple fix would be to use the common `object` type on the channel, but that would require all others to change the type of their channel as well, forfeiting all benefits of having a strongly typed channel, simply to use it in a mixed-type operation. Instead, this is implemented using an instance implementing the interface `IMultisetRequestUntyped` for each request. By having the ordinary, typed, `MultisetRequest` instances implement this interface, the usage is the same from the user perspective, as shown in listing 2 where the update channel handles strings. While the two function calls look similar, the call to `ReadOrWriteAnyAsync` is passing a list of untyped requests. To invoke the actual typed channel operations, the implementation uses *reflection* to extract the target method from each channel, and invokes the corresponding method. This adds a minor overhead to the function call, as each channel type needs to be resolved, but is otherwise comparable to the same-type version in terms of performance.

```
var cur = 0, prev = 0;
var updatechan = ChannelManager.GetChannel<int>("update");
var curchan = ChannelManager.GetChannel<int>("cur");
var prevchan = ChannelManager.GetChannel<int>("prev");

while(true) {

  var res = await new [] {
    MultisetRequest.Read(updatechan),
    MultisetRequest.Write(cur, curchan),
    MultisetRequest.Write(prev, prevchan),
  }.ReadOrWriteAnyAsync();

  if (res.IsRead) {
    prev = cur;
    cur = res.Value;
  }
}
```

Listing 1. Example for `ReadOrWriteAny`.

```
var cur = 0, prev = 0;
var updatechan = ChannelManager.GetChannel<string>("update");
var curchan = ChannelManager.GetChannel<int>("cur");
var prevchan = ChannelManager.GetChannel<int>("prev");

while(true) {

  var res = await new [] {
    MultisetRequest.Read(updatechan),
    MultisetRequest.Write(cur, curchan),
    MultisetRequest.Write(prev, prevchan),
  }.ReadOrWriteAnyAsync();

  if (res.IsRead) {
    prev = cur;
    cur = int.Parse((string)res.Value);
  }
}
```

Listing 2. Example for `ReadOrWriteAny` with mixed types.

1.2. Overflow Strategies

In an unbalanced network, it is easy to create situations where readers are overwhelmed with requests. If the writers are blocked when writing to the channel, then that will naturally prevent this problem. However, it may be desirable for the writers to be able to continuously write values to a channel without having to wait for the channel to be available. This situation could be desirable if a process is reading a sensor and continuously writing the current value to a channel. If an unbuffered channel is used this means that the sensor will not be read while the process is writing to the channel. Adding a buffer to the channel gives some flexibility, such that the reader and writer can work *out-of-sync*, but will not help if the writer is faster than the reader, as that will just give a brief time to fill the buffer, and then revert to the unbuffered scenario. Consider the example in listing 3 where the reader needs to alert a shutdown mechanism if the value is too high, but otherwise report the current value to a

reader. If the `report.Write()` call is blocking because the reader is busy, the sensor is not read, and hence the warn signal is not reported if the sensor value increases. One could argue that the problem could be solved with an alternation, but in this particular example, the sensor is a hardware unit and thus not a channel input that can be part of an alternation.

```
var warn = ChannelManager.GetChannel<int>("warn");
var report = ChannelManager.GetChannel<int>("report");

while(true) {

  var current = ReadHardwareSensor();
  if (current > 10)
    warn.Write(current);
  report.Write(current);
}
```

Listing 3. Example with a sensor performing blocking writes.

```
var warn = ChannelManager.GetChannel<int>("warn");
var report = ChannelManager.GetChannel<int>("report",
  maxPendingWriters: 1,
  pendingWritersOverflowStrategy:
    QueueOverflowStrategy.FIFO
);

while(true) {

  var current = ReadHardwareSensor();
  if (current > 10)
    warn.Write(current);
  report.WriteNoWait(current);
}
```

Listing 4. Example with a sensor performing non-blocking writes.

In such a scenario it might also be sensible to design the reader process such that it does not require all values, but simply uses the most recent value. In other words, old sensor values can be discarded. To solve these issues, we added the option to set a maximum queue length on a channel and also set a strategy that determines what will happen once the channel is overflown. Note that this is different from a buffered channel, as a buffered channel is semantically equivalent to a series of processes that forward a value. The queue in the channel is a list of pending, or blocked, operations, whereas the buffer in a channel is a list of writes that have been processed, but not yet collected by a reader.

The JCSP `OverFlowingBuffer` and `OverWritingBuffer` channels serve a similar purpose, but implement this differently as they always signal success to the writer. In CoCoL, an overflowing write action will be cancelled, which is signalled to the writer as a special exception state, allowing the writer to detect lost messages. Write operations that are buffered return success to the writer, and can only be lost in the case of posioning the channel, which is equivalent to poisoning a buffer process.

The available strategies for handling overflow in a CoCoL channel are `FIFO`, `LIFO`, and `Reject`. The `FIFO` strategy will expire items that have been the longest in the queue, the `LIFO` strategy will expire items that have been the shortest time in the queue, and the `Reject` strategy will reject new items from being inserted while the queue is filled. For the sensor

example we can simply add a maximum size of pending writers and the queue strategy to the channel as shown in listing 4.

This change will allow the `report.WriteNoWait()` call to always proceed, expiring old values if needed. The reader will then always receive the latest value, even if the writer is faster than the reader. Note that without the limitations on the channel, the calls to `report.WriteNoWait()` would cause the list of pending writes to grow until the machine is out of memory, and would cause the reader to process a backlog of values instead of just the most current. The performance overhead for discarding the extra entries, is limited to removing items from the list.

For a slightly different scenario, where the sensor reader would like to know that the reader is not able to handle a particular request, the strategy `Reject` could be used, such that the call to `report.Write()` gives an exception, or the `Task` value from the previous write could be stored and examined.

1.3. Support for the Portable Class Libraries

As the world, particularly software developers, is focusing on mobile *apps*, the CLR, C#, and the related eco-system is also targeting *app* development. Since the devices running such apps are usually limited in terms of memory and processing power, Microsoft has defined a reduced version of the BCL[2] known as the Portable Class Libraries, PCL, for use on such devices.

We have added support for using CoCoL with the PCL, such that apps can benefit from the CSP programming model as well. Most of the functionality used by CoCoL is directly supported in the PCL, with the exception of some reflection functionality and a sorted list.

The channel expiration mechanism is implemented with a sorted list, such that a single timer keeps track of all pending channel timeouts, rather than instantiating a timer for each call [1]. Since a sorted list is rarely used, it has been omitted from the PCL and we have simply added our own implementation which keeps the list sorted in $O(lg\ n)$.

The reflection functionality in PCL is reduced, both to save space, but also to ensure that it can be used with runtimes that have limited capabilities. Fortunately, CoCoL uses only a limited amount of reflection, to deal with mixed types as described in section 1.1, which we have adapted to work correctly with both BCL and PCL.

1.4. Channel and Execution Scopes

One of the attractive features of a CSP-based programming model is the ability to encapsulate components, which decreases complexity and increases reusability. It is possible to create both named and unnamed channels in CoCoL, such that a channel reference can be acquired either by name or by reference. The benefit from using named channels is that an embedded string is enough to access the channel, as opposed to passing an object reference through multiple layers of processes.

However, when creating reusable encapsulated components, using named channels is difficult, as the names used in the component may collide with the names used by the calling program. One common solution to this problem is to introduce namespaces, which can be implemented, i.e. by requiring the user to prefix channel names with a unique string. But the namespace approach fails when the user instantiates two instances of an encapsulated component, say two fibonaci generators, as they will have the same prefix.

Rather than providing an elaborate system to provide extra prefixes to prefixes, CoCoL implements *scopes*, such that a channel always belongs to a scope. An encapsulating com-

[2]Base Class Libraries, the libraries used by all CLR programs, containing lists, hashtables, filesystem, sockets, etc

ponent can then generate a new scope, to which all the channels it creates are associated. A `ChannelScope` *inherits* the channels in its parent scope, but can override a named channel with a local channel. For fully encapsulated components, an `IsolatedScope` is also provided, which stops the inheritance at the scope. Rather than explicitly passing the channel scope as a reference, we utilize the `ThreadLocalStorage` feature in CLR to attach the channel scope to the caller. This approach ensures that a process can instantiate channels without any knowledge of what scope it belongs to, thus enabling fully isolated components. The scope usage is shown in listing 5, where two different channels are named `out`. The scope isolates this, such that the two `async` tasks, can request a shared channel named `out` without seeing the external channel with the same name. By using a reference to the channel from the outer scope, the inner network can even communicate with the outside network, as illustrated in listing 5.

A similar approach is used to implement an `ExecutionScope`, which defines how to execute tasks once they are ready. An `ExecutionScope` can be used for fine grained control over a subset of the network, for example to use a custom threadpool which limits the number of active threads.

```
var external =
  ChannelManager.GetChannel<int>("out");

using(new IsolatedScope())
  await Task.WhenAll(

    Task.Run(async () => { // Process 1
      var internal =
         ChannelManager.GetChannel<int>("out");

      while(true)
        await internal.WriteAsync(42);

    }),

    Task.Run(async () => { // Process 2
      var internal =
         ChannelManager.GetChannel<int>("out");

      while(true) {
        await external.WriteAsync(
          await internal.ReadAsync()
        );
      }
    })
  );
```

Listing 5. An example of an isolated component.

1.5. Automatic Channel Wiring

When constructing a CSP-style network, one must *attach* channel ends to the correct processes, which is also known as *wiring* the processes. Consider the code example in Figure 6, which is taken from the PyCSP paper [7]. In the example, the `producer` outputs tasks read by a `worker`, which is then collected by a `consumer` through a `barrier`[3]. The equivalent

[3]The code example is taken verbatim from the paper, but can be written more concisely with the current PyCSP library

code in CoCoL could be written as in listing 7, where the `AsRead` and `AsWrite` suffixes are optional, but included for similarity with the PyCSP example.

```
feeder = One2AnyChannel()
collector = Any2OneChannel()
done = Any2OneChannel()

Parallel(
  Process(producer, protein, map, place, feeder.write),
  Process(worker, feeder.read, collector.write, done.write),
  Process(worker, feeder.read, collector.write, done.write),
  Process(worker, feeder.read, collector.write, done.write),
  Process(worker, feeder.read, collector.write, done.write),
  Process(worker, feeder.read, collector.write, done.write),
  Process(barrier, 5, done.read, collector.write),
  Process(consumer, collector.read))
```

Listing 6. PyCSP example for a prototein network.

```
var feeder = ChannelManager.GetChannel<int>();
var collector = ChannelManager.GetChannel<int>();
var done = ChannelManager.GetChannel<bool>();

Task.WhenAll(
  Producer(protein, map, place, feeder.AsWrite()),
  Worker(feeder.AsRead(), collector.AsWrite(), done.AsWrite()),
  Worker(feeder.AsRead(), collector.AsWrite(), done.AsWrite()),
  Worker(feeder.AsRead(), collector.AsWrite(), done.AsWrite()),
  Worker(feeder.AsRead(), collector.AsWrite(), done.AsWrite()),
  Worker(feeder.AsRead(), collector.AsWrite(), done.AsWrite()),
  Barrier(5, done.AsRead(), collector.AsWrite()),
  Consumer(collector.AsRead()));
```

Listing 7. CoCoL example for a prototein network.

```
public class Worker : ProcessHelper {
  [ChannelName("feeder")]
  private IReadChannel<int> feeder;

  [ChannelName("collector")]
  private IWriteChannel<int> collector;

  [ChannelName("done")]
  private IWriteChannel<bool> done;

  ... code omitted ...
}

Task.WhenAll(
  new Producer(protein, map, place),
  new Worker(), new Worker(), new Worker(),
  new Worker(), new Worker(),
  new Barrier(5), new Consumer());
```

Listing 8. CoCoL example for a worker process.

Instead of passing the channel references to the method, we can define the channels inside the method and thus avoid cluttering the program with channel references. This approach is shown in listing 8, where a `Worker` method has fields that are *annotated* with the name of the channels they reference, and inherits from the `ProcessHelper` class that performs the wiring. Combined with the channel scopes explained in 1.4 this makes it simpler to define encapsulated components, where each component defines what channels it uses, rather than relying on an external process to perform the wiring. While not shown in the example, the implementation also allows the user to specify the channel properties in the annotations, with regards to overflow and buffering as explained in section 1.2.

1.6. Leave and Join

Revisiting the code examples in listing 6 and 8, we can see that the injected `Barrier` process is required to avoid a race condition that is likely to happen if the `Worker` processes are not computing with the same speed. If the `Barrier` was not present, the first worker to finish would poison the channel, causing the remaining worker results to be lost when they attempt to write. While it is trivial to solve this problem with a barrier-like process, it is very likely that novice programmers would hit this particular issue, yielding a faulty program. A solution to this was first mentioned in *PyCSP Revisited* [8], where the authors suggest using the terms *leave* and *join* to implement a counting technique to delay the shutdown on a channel until either all readers or all writers have *left* the channel. This approach solves an issue similar to the one described in *JCSP-Poison: Safe Termination of CSP Process Networks* [9], but by using a counter to delay poisoning, rather than relying on the programmer to choose the correct poison method.

The PyCSP idea was added to CoCoL, retaining the names `Join` and `Leave` for joining and leaving a channel respectively. Unlike the PyCSP implementation, the channel is marked as poisoned after all readers or all writers have called `Leave`[4]. The same mechanism that is used to automatically wire up the channels in a class, will also automatically call `Join` and `Leave` during startup and shutdown respectively, using the channel type to determine if the channel is joined as either a reader or a writer. This implementation means that we can remove the `Barrier` process, as well as the `done` channel in Figure 8, without introducing any shutdown hazards.

The implementation of the `Join` and `Leave` process changes a small amount of code in the core `Channel` class, by keeping two integer values per channel instance, one for the readers, and one for the writers. Calling `Join` or `Leave` simply adjusts the read or write counter and calls `Retire` once the counter reaches zero.

1.7. Isolating Processes

Functional programming is touted as a solution to many problems with programming [10], and interestingly the arguments and solutions have many resemblances to CSP. We leave the general comparison of CSP and functional programming to others [11,12], but focus on the concept of functions in functional programs as being *side effect free*. Working with such functions makes it easier to reason about the results, as they can be used in a compositional manner. In a CSP context, an equivalent idea is that each process is encapsulated and also used to build composite networks. As CoCoL is implemented in a non-functional programming language, we cannot make the guarantee that a process is *side effect free*, but we can use language features to set up an encapsulating scope that encourages use of local state only.

[4]PyCSP uses the name `ChannelRetireException` and distinguishes between *poisoned* and *retired* states, where CoCoL only has a single poisoned state which is called *retired* and throws a `RetiredException`

This is not directly equivalent to having a *side effect free* function, but by using only local variables we can encapsulate the state to be only in the process.

To achieve this encapsulation, we utilize three different language techniques: lambda functions, variable scopes, and anonymous types. Using these three features in combination, we can write the prototein network from Figure 8 in a simpler way, as illustrated in Figure 9. In this example, the `Worker` process is not a class deriving from `Process`, but instead implemented as a static method. This ensures that the process can only read and write local variables, as well as variables passed to the method, and not access a `this` pointer. Inside the method, a new *anonymous type* is created, which allows the user to give a local name to the channel, as well as specify the type and intended usage for the channel. The instance is then passed to the lambda function with the name `self`, and it can then be accessed in a fully type safe manner. The `RunTask` method looks up all desired channels, calls the `Join` method, and assigns them to the instance. The lambda method is executed within a `try/catch` block, and automatically calls `Leave` on all channels once the process quits.

```
private static Task Worker() {
  return AutomationExtensions.RunTask(
    new {
      feed = ChannelMarker.ForRead<int>("feeder"),
      coll = ChannelMarker.ForWrite<int>("collector")
    },

    async self => {
        while (true) {
          var data = await self.feed.ReadAsync();
          ... code omitted ...
          await self.coll.WriteAsync(result);
        }
      }
    );
}

Task.WhenAll(
  Producer(protein, map, place),
  Worker(), Worker(), Worker(), Worker(), Worker(),
  Consumer());
```

Listing 9. CoCoL example with closures.

2. Network Support

One of the benefits of a channel based communication model is the ability to use different communication mechanisms without changing the channel abstraction. This ability makes it easy to utilize a network connection and build a distributed system. From the users perspective, the channels work the same, although they might become a bit slower.

This approach has been implemented in other CSP libraries, like JCSP [3], CPPCSP [4] and PyCSP [7]. From a users perspective, a network channel is an explicit choice in JCSP and CPPCSP, where the user needs to instantiate a special network channel instead of the normal channel types. In PyCSP, the channels do not have a specific type, but instead *upgrade* their type based on the type of communication they participate in.

As argued for PyCSP [13], it is desireable to have a single channel from a usability perspective, but often desireable to have different implementations from a performance perspective.

2.1. Connecting Channel Ends

In CoCoL we have implemented a network channel type as a distinct type, like it is implemented in JCSP and CPPCSP. To avoid having the user instantiate a specific channel type, we instead implement a channel scope, as described in section 1.4, that will choose a channel type based on the channel name. This restricts the use of network enabled channels to named channels, but that is not a real limitation as it would not be possible for two different processes to share a reference to a channel, as they do not share memory space. An example implementation could choose to prefix network channel names with `net` , and then setup the network channel scope to create all channels named `net *` as network channels.

The implementation relies on a single *channel server*, which hosts a normal CoCoL channel, and provides an interface to the channel via TCP, much like the solution described for PyCSP [13]. CoCoL does not yet support multiple channel servers, nor the *name server* found in JCSP, as we want to use the current implementation as an evaluation platform before we decide on the future implementation. It does not support mobile channel ends, nor moving channel ends or channgel servers in the current implementation. The network implementation is shown in Figure 1.

The *channel server* uses a very simple protocol, implemented over TCP, where it listens to incoming requests and forwards channel responses to the TCP connection. The protocol comprises a header and an optional payload. The header contains a unique request ID for each requests, such that the response can later be paired, as well as the message type and a few additional bookeeping items. A typical message is just over 200 bytes without any payload data. The header and the payload are both serialized as a length prefixed JSON entry, and only data that can be serialized with JSON is currently supported, similar to the limitation imposed by JCSP. The network implementation is shown in Figure 1. The *network client*, *network client handler*, and *network server handler* in Figure 1 are implemented as processes using local CoCoL channels to forwards messages in the correct order. The *network client* utilizes `await` statements, such that it can scale to a large number of concurrent requests.

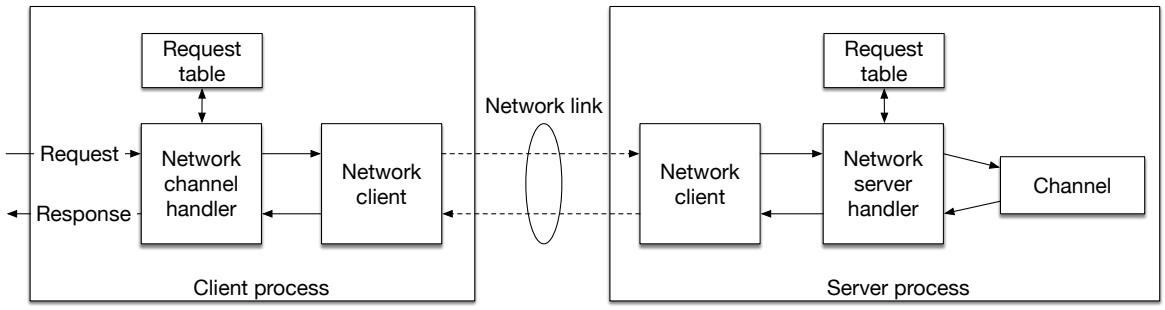

Figure 1. Overview of the components involved in handling a network request.

2.2. Reads and Writes

The two core operations in CoCoL are *read* and *write*, so naturally there are messages for these operations, namely `ReadRequest`, `ReadResponse`, `WriteRequest`, and `WriteResponse`. On the client side, the channel is simply implemented such that it generates a unique request ID for a read or write and then sends a package to the *channel server* and awaits a response. When the *channel server* receives a message, it locates the requested channel and issues the corresponding operation on the channel.

When the operation succeeds, the *channel server* will then respond the message back to the initiator. This simple setup ensures that all operations are performed within a single process, the *channel server*, and the client merely acts as a proxy when transmitting the

messages. Similar messages are supported for other channel operations, like *JoinRequest*, *LeaveRequest* and *RetireRequest*, as well as negative results like *RetiredResponse*, *ErrorResponse*, and *TimeoutResponse*. All message types are listed in Table 1.

2.3. Timeouts

Even though the channel is now distributed, it can still be used like any other channel. However, since properties are evaluated on the *channel server*, there is a delay between operations, meaning that poisoning a channel will not be immediately visible as with a local channel, but will otherwise work the same.

The channels still guarantee that messages are handled in the order they arrive, but due to network characteristics, there can now be varying latencies, meaning that the order of arriving messages can change. Timeouts are also handled on the *channel server*, so they can happen slightly later than what the user requests. A timeout is stored as an absolute time once recorded, such that any delay in transmission is not added to the supplied maximum wait time. If the requests arrives after the specified timeout, it is treated as a probing call, similar to a *skip guard*.

2.4. Alternation

As *external choice* in CoCoL is implemented with *two-phase commit*, we have implemented a proxy for the `ITwoPhaseCommit` that simply forwards the `OfferRequest` and waits for either a `CommitResponse` or `WithdrawResponse`, as described in Table 1. This allows a client full freedom in mixing and matching channels, datatypes, reads, and writes in an alternation statement. The *channel server* simply sends the messages to the client, which can then host an unmodified `ITwoPhaseCommit` instance. For requests that do not utilize `ITwoPhaseCommit`, i.e. requests that are not part of an alternation statement, a flag is sent to the *channel server*, such that it avoids sending these extra messages.

The implementation of the proxy for network based `ITwoPhaseCommit` lead to a minor change in channels. Previously, all calls to `ITwoPhaseCommit` were expected to return immediately, which was reasonable, given that they only needed to query a single variable. With the network support, it can suddenly take several milliseconds or more before a response is received. In the previous design, this would cause the channel to block all threads calling the channel, until a response had been processed.

To work around this, we changed the implementation of `ITwoPhaseCommit` to use `async` logic for all calls. Via a lock-like construct, dubbed an `AsyncLock`, we can retain the code used previously, without blocking threads. If the `AsyncLock` is not taken when calling, the thread making the request is forwarded, thus reducing the overhead of the lock to checking a variable on entry for non-competing calls on a channel. Should the `AsyncLock` be held, the call is queued in dispatch queue, similar to the queues used inside the channel to keep track of pending readers and writers. This queue mechanism ensures that the channel retains the order, as they are passed into the channel queues.

With the `ITwoPhaseCommit` abstraction, we can support all types of alternation, such that the decision for choosing a channel is kept in the process that performs the request, thus removing the need for communication between the channels involved in an operation. The simple *request-response* scheme for *two-phase commit* is implemented in the same way as normal requests, so we re-use the setup illustrated in Figure 1 to pass all messages listed in Table 1.

2.5. Overhead and Latency

With intra-process communication it is possible to merely pass a pointer or reference, As two different processes do not share a common memory space, communication between two

Table 1. Network message types.

Name	Description	Sent from
ReadRequest	Intent to read	Client
WriteRequest	Intent to write	Client
RetireRequest	Intent to retire	Client
JoinRequest	Intent to join	Client
LeaveRequest	Intent to leave	Client
ReadResponse	Completed read	Server
WriteResponse	Completed write	Server
TimeoutResponse	Timeout on read or write	Server
CancelResponse	Cancel on read or write	Server
RetiredResponse	Channel was retired on read or write	Server
FailResponse	Read or write failed	Server
CreateChannelRequest	Initialize channel, includes channel options	Client
OfferRequest	The offer phase of two-phase commit	Server
OfferAcceptResponse	The two-phase offer was accepted	Client
OfferDeclineResponse	The two-phase offer was rejected	Client
OfferWithdrawRequest	The two-phase transaction is rolled back	Server
OfferCommitRequest	The two-phase transaction is committed	Server

processes adds processing overhead. If the communication method involves travelling over a network link, there will be a latency overhead as well.

One could attempt to use buffered channels, in an attempt to hide the latency overhead, but this will not have the desired effect. Firstly, the buffering only affects the writing side, as the logic for a buffered channel simply accepts write requests even when there are no matching readers. Secondly, the buffering is conceptually the same as writing to a channel, thus the write call needs to be transported to the channel server and back, inducing delays, but it will likely not wait in the queue on the channel server.

There is no real solution to this problem, if we want to retain all the properties of channels, as we cannot determine if the channel will reject or accept a request ahead of time. If we accept that a few messages can be *in-flight* when the channel is poisoned, we can implement a windowing approach, where we accept write calls, and forward them, but do not delay the writer. Internally, all *in-flight* requests can be stored in a queue, and they will work the same as normal channel interactions, and thus work correctly with respect to timeouts and alternation. When the channel is poisoned, the writer is notified of this change, *after* the channel has been shut down, meaning that some writes are seen as successful by the writer, but they actually failed.

The reader on a network channel experiences the exact same delay, but we cannot use *in-flight* requests in the same manner as writes, because the value being read from the channel is not known in advance. Instead we can ensure that we issue a number of preemptive read requests on a channel, and keep these in a queue. When the calling program attempts to read, we emit a new pending read, add it to the queue, and return the oldest pending read. This provides the same kind of windowing behavior as the write method, but since we emit the read request ahead of time, we cannot register the correct two-phase offer when the read is emitted. For this reason, it is not possible to use alternation with these *windowed* requests. Likewise, the timeout value is not known in advance of the actual call, and is not supported either. Similar to the approach for write messages, there can be read requests *in-flight* when the channel is poisoned. However, if the reader keeps reading until discovering that the channel is poisoned, no messages will be lost, as the oldest reads are processed first.

Even though the implementation for the write requests is different from a traditional

buffer, this is semantically equivalent to sequence of processes that forward a value. In such a construction, a write in one end of the chain will report as successful, but poisoning the output of the sequence, will cause all messages inside the chain to be lost. Similarly, the read requests can also be modeled as a sequence of processes, which will carry a poison signal from the input to the output, and will cause lost messages if the reader stops reading before the poison is detected.

The trade-off in speed versus correctness requires that the programmer is well informed of the consequences, but can work well for typical *worker pool* setups. The authors are not aware of other CSP implementations that implements *optimistic* read ahead.

3. Experiments

All of the enhancements described in section 1 are all added on top of the implementation, and only serve to make it simpler to write correct programs. We have leveraged that functionality in the example programs, but have yet to evaluate the impact on users.

For the network based channels, we are interested in determining the amount of additional overhead and latency involved with passing messages over a network channel. As the implementation contains performance enhancements for channels that are not part of an alternation, we would like to test both types. To reduce the amount of jitter and latency noise we would expect from a network, we have set up some tests to communicate over the loopback adapter. This ensures that we measure the overhead from serialization and deserialization, as well as any TCP induced overhead, but we do not have any additional latency, as the data never leaves the machine. To measure a more realistic scenario, we are also measuring the performance, using a *channel server* hosted on Amazon EC2.

The experiments are performed on a MacBook Pro with a 2.8 Ghz Intel i7, running OSX 10.11.5 and a 64bit Mono version 4.2.1.102. The Amazon EC2 server is running Ubuntu 14.04.3 LTS on an Amazon t2.micro instance located in the eu-west-1 region, and has Mono 4.2.3 installed. During the experiments, the round-trip time to the Amazon server was measured continuously and remained around 45 milliseconds, with minor spikes measuring up to 2000 milliseconds.

The measurements presented here only compare one setup of CoCoL with another, and this shows significant performance differences. We have not performed compareable measurements with existing libraries, such as JCSP and PyCSP, as the current CoCoL network implementation is too rudimentary to be compared to the more feature complete libraries.

3.1. CommsTime

For the first experiment, we use the classical CSP benchmark CommsTime. In CommsTime, a single data item, is passed around in a ring with a *delta* process both forwarding the data, and simultaneously emitting a value that can be measured in an external process and thus time the network. To further measure the overhead in the network, and compare it to the overhead with local channels, we measure a baseline setup where no channels are network-based. On the test machine, we measure this communication time to be 33 microseconds per channel communication.

3.1.1. Commstime With a Single Network Channel

To measure the overhead introduced by the network channel implementation, we use a setup, where one channel in the *delta* process, the one emitting to the counter, is network based. We first run this setup using a locally hosted *channel server*, communication over the loopback adapter, and obtain the results in Figure 2. For this run, we can conclude that the computation overhead is rather large, 444 microseconds, which is approximately 15 times larger than the

non-network based channel. As the buffers do not influence the communcation time, we can conclude that the overhead is not caused by the communication latency, but is caused by computation.

We then repeat the same setup, but with a channel server hosted on Amazon EC2. As shown in Figure 3, and as expected, the overhead for this is much larger, 33100 microseconds, as we now need to add the network latency as well. However, since the latency is essentially idle time, we can utilize the wait time better by keeping *in-flight* requests as explained in section 2.5. At the measured 45 milliseconds round-trip time, we obtain a linear speedup by increasing the buffer size until we have around 100 *in-flight* requests. At this point we are down to a communication overhead of 225 microseconds, or approximately 13 times slower than the non-network based channel. Interestingly, this is even faster than the setup where we host the channel server locally. This can be explained by the fact that parts of the overhead is in serializing and deserializing the packages, which is now performed on two different machines. During the tests we also observed the CPU utilization on both the local and the remote machine and observed an increasing CPU usage on both as we increase the buffer size. With the measurements for buffer sizes 100 and 200, we observed that the remote server was utilizing 100% of the CPU resources, indicating that we could possibly obtain better results if we upgrade the channel server.

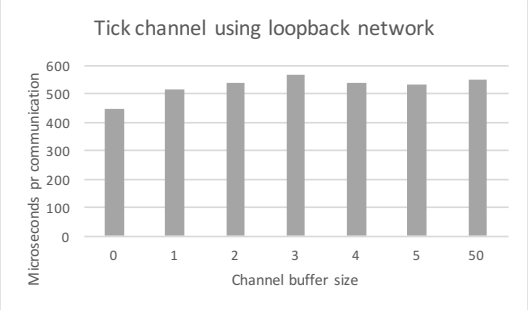

Figure 2. Communication time for a single network-based channel with a channel server on the same machine.

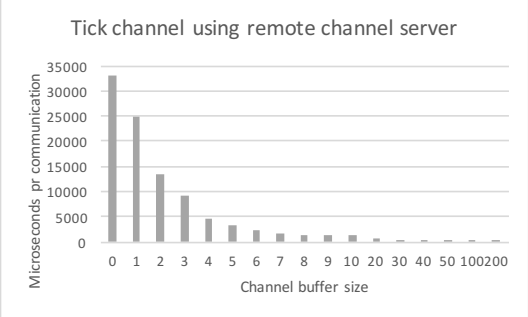

Figure 3. Communication time for a single network-based channel with a channel server on another machine.

3.1.2. CommsTime With Multiple Network Channels

To further explore the overhead in the network implementation, we ran the CommsTime benchmark with all channels being network channels. For the first setup, shown in Figure 4, we used a channel server that was hosted locally. For the second setup, shown in Figure 5, we used the channel server hosted at Amazon. We observe that we obtain a significantly higher communication time, 1632 microseconds and 104100 microseconds for local and remote respectively, corresponding to approximately 50 and 3000 times slower than the non-network channel. As we expect from the results with only a single network channel, we cannot obtain speedup with buffering in the case where the channel server is hosted locally. Interestingly, we can see that it is not possible to obtain speedups in the case where the channel server is placed on a remote server either. This happens because there it is not possible to progress, as each process in the ring depends on the previous. When it is only the tick channel that is network based, the ring can continue to communicate and it will write messages to the tick channel without waiting for completion, but when all processes are pending, we cannot send write messages faster, as the ring is blocked.

3.1.3. CommsTime With Alternation

To evaluate the effect of the *two-phase commit* overhead, we repeat the CommsTime experiment where the measurement channel is network based, but add an artificial two-phase in-

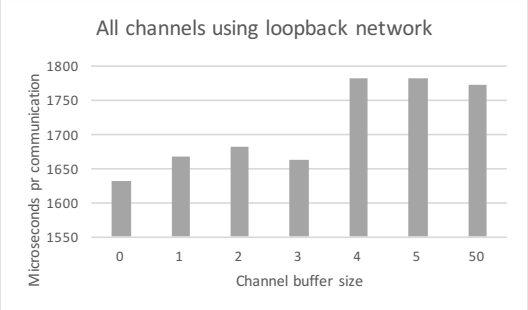

Figure 4. Communication time for all network-based channels with a channel server on the same machine.

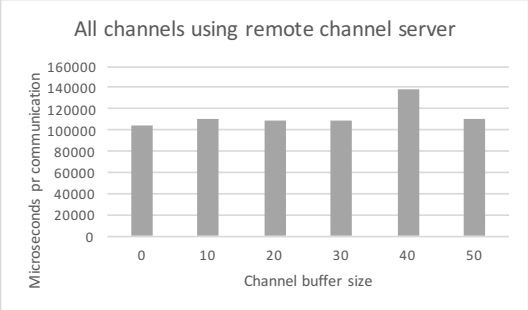

Figure 5. Communication time for all network-based channels with a channel server on another machine.

Table 2. Communication time for an 8x4 fair alternation.

Setup	Microseconds pr read	Overhead factor
Non-networked	75	1
Local channelserver	15258	203
Remote channelserver	120193	1603

stance when reading from the channel. Again we vary the buffer size to evaluate how much of the latency overhead can be hidden. By comparing Figure 2 with Figure 6 we can observe that the alternation doubles the delay, which is to be expected as the two-phase commit process sends a separate request/response pair. In Figure 7 we can observe that this extra communication also prevents the write buffer from achieving the same speedup as we observed in Figure 3.

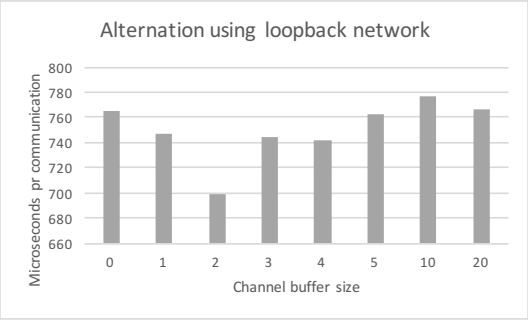

Figure 6. Communication time for alternation with a channel server on the same machine.

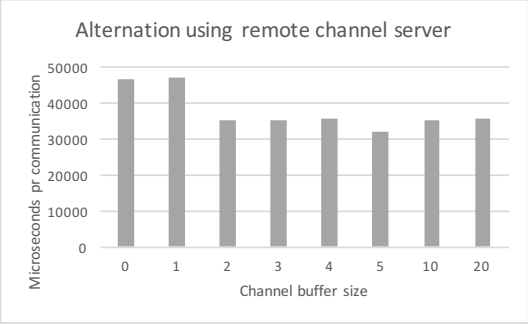

Figure 7. Communication time for alternation with a channel server on another machine.

3.2. StressedAlt

Another classic CSP benchmark is *StressedAlt*, where a single reader is bombarded with write requests from a set of writers. To further complicate matters, the reader uses *fair alternation* meaning that it will choose which of the potential channels to read in a round-robin fashion. This means that all reads use a shared *two-phase commit* to ensure that the reads are performed in a *fair* manner. We ran the example with the shared channel as a non-networked version, as well as with a local channel server and a remote. With fair alternation, we are not able to use buffers, because we do not know which channel is the next *fair* choice, until after we have read. The results for running a small example with 8 channels and 4 writers are shown in Table 2, and indicates that there is a large amount of overhead involved in performing priority alternation over a network channel.

3.3. Mandelbrot

For the Mandelbrot benchmark we use a simple approach, where a process is spawned for each pixel in the resulting image. This has the benefit that each process will attempt to write the shared result channel, and thus create the same situation as would be provided by the buffer approach described in section 2.5, without explicitly designing for it. The reading end, on the other hand, is equipped with a buffer that reads ahead of time, to hide the latency. Consistent with the other results, we can observe in Figure 8 that it is not possible to hide the processing overhead with buffers when we are using a local server. As we observed in Figure 3, it is possible to hide some latency when the channel is on a remote server, and we can observe the same tendency in Figure 9. If we run the same setup with non-networked channels, we obtain that each pixel takes 55 microseconds to compute, and we can compare to the best result for a local channel, 2007 microseconds, and the best result for a remote channel, 1019 microseconds. Again, the remote server handles half of the overhead related to serialization and deserialization, and thus the remote setup is faster when we hide the communication latency with buffers.

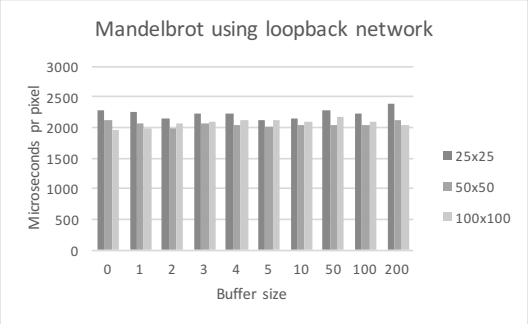

Figure 8. Communication time for Mandelbrot rendering with a channel server on the same machine.

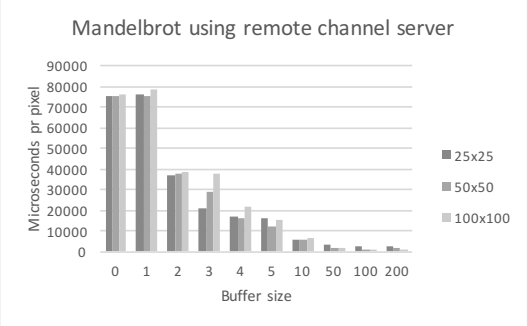

Figure 9. Communication time for Mandelbrot rendering with a channel server on another machine.

4. Future Work

From the results presented in section 3, we know that some setups can execute efficiently over a network, but requires the use of buffers. We would like to add additional features, that enable the programmer to achieve such results in more setups. Beyond that, the network channel implementation is lacking many desirable features.

4.1. Error Handling

In the current implementation there is no handling of network errors; we expect the channel server to be available and responding, and assume that the clients do not crash or drop the connection. Supporting such scenarios is not an infeasible task, but since we cannot distinguish between a crash or a network outage, we need a clear definition of what the correct response to a failed communication event would be. A similar mechanism could be used to provide timeout handling in cases where the client or server is not responding.

4.2. Name Server

In more realistic scenarios than the ones we experimented with here, it is very likely that the channel server will be a bottleneck, impeding overall performance of a distributed system.

To solve this, we intend to add a *name server*, similar to the one found in JCSP. The name server will be responsible for choosing which channel server will host a given channel. Each channel server can report usage hints back to the name server, such that channels can be re-assigned to other channel servers if that appears to be beneficial.

This multi-tier setup makes it easier to scale to very large installations, as the name server only serves an advisory role after the network has been established, and the channel server instances can be scaled to fit the load. For very large instances, the name server can also be distributed, provided that the name server instances can guarantee that the same channel is not assigned to multiple hosts.

4.3. Transaction Logs

To provide resilience in the face of errors, we would like to investigate use of transaction logs to allow repeats of failed operations. The general idea is to write to a persistent medium prior to sending a message onto the network, and then write again after the message has been sent. Should the instance crash during the operation, it is possible to read the transaction log, and see which messages should be retransmitted. The recipient then needs to handle receiving the same message twice, and the system would be resilient to crashes, provided that the transaction logs are stored on durable media.

With transaction logs, it should be possible to start a process, inject messages, shut down the process, and at some later time start the process and resume participation in the network.

5. Conclusion

In this paper we have described the most recent additions to CoCoL, comprising a number of features for writing more concise, correct and encapsulated programs, as well as the addition of a basic network-enabled channel. The experiments we present show that the *two-phase commit* approach to alternation is a viable approach, but requires some work to achieve optimal performance.

All source code, including the benchmarks, can be found on the project website [14].

Acknowledgements

The authors would like to thank the anonymous reviewers, as well as the editors, for their helpful comments

References

[1] Kenneth Skovhede and Brian Vinter. CoCoL: Concurrent communications library. In *Communicating Process Architectures*, 2015.

[2] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[3] Peter H Welch, Neil CC Brown, James Moores, Kevin Chalmers, and Bernhard HC Sputh. Integrating and extending jcsp. *Communicating Process Architectures*, 65:349–370, 2007.

[4] 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*, pages 183–205, jul 2007.

[5] Jan B. Pedersen and Andreas Stefik. Towards millions of processes on the JVM. In *Communicating Process Architectures*, 2014.

[6] Butler Lampson and Howard Sturgis. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California, 1979.

[7] John Markus Bjørndalen, Brian Vinter, and Otto J Anshus. PyCSP-communicating sequential processes for python. In *Communicating Process Architectures*, pages 229–248, 2007.
[8] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP revisited. In *Communicating Process Architectures*, pages 263–276, 2009.
[9] Bernhard HC Sputh and Alastair R Allen. JCSP-Poison: Safe termination of CSP process networks. In *Communicating Process Architectures*, volume 63, pages 71–107, 2005.
[10] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.
[11] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
[12] Charles Antony Richard Hoare and He Jifeng. *Unifying theories of programming*, volume 14. Prentice Hall Englewood Cliffs, 1998.
[13] Rune Møllegaard Friborg and Brian Vinter. Verification of a dynamic channel model using the SPIN model checker. In *Communicating Process Architectures*, pages 35–54, 2011.
[14] K. Skovhede. Cocol source code. `https://github.com/kenkendk/cocol`. [Online; accessed June 2016].