Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

C++CSP Networked

Neil BROWN
QinetiQ, Malvern Technology Centre, St Andrews Road,
Malvern, Worcestershire, WR14 3PS, United Kingdom
neil@twistedsquare.com / ncbrown@QinetiQ.com

Abstract. C++CSP is a library for C++ enabling direct implementation of CSP concurrency design. It provides an extended set of CSP primitives that follows the model captured by occam and JCSP, with an API similar to the latter. It runs on most platforms, with efficient realisation for both Windows and Unix/Linux. It was released under the open source Lesser GNU Public Licence in January, 2004. That version supports only concurrency within a single machine. This paper details the development of a network capability for the library and reports some benchmarks, which are encouragingly fast. The design of C++CSP networking follows several of the decisions made for the JCSP Networking Edition (e.g. naming and automatic multiplexing, though not yet a Channel Name Service). Three further extensions to the library (channel poisoning, factories and bundles) are also presented.

1 Background

Communicating Sequential Processes (CSP) [1, 2] is a concurrency model centred around concurrently executing processes that communicate using channels. CSP channels are point-to-point; processes communicate through writing or reading on *channel ends*, rather than the writer having the address of its intended reader. CSP communications are also synchronised. When a channel communication takes place, both parties must engage. When a writer has finished its communication, it knows that its reader has read its message (rather than asynchronous communication, where the writer has no such information). Crucially, a process knows that its internal state cannot be changed behind its back, by being updated through the arrival of messages outside its control.

An advantage of CSP is that the processes do not have to know anything of the delivery mechanism of the channels they are using. Provided that the channels follow CSP channel semantics, their implementation details (e.g. routing) are irrelevant to the processes using them. This means that the channels can be network transparent; two processes will function the same way whether they are communicating using *internal* (on the same machine) channels or over *networked* (between machines) channels.

Due to the limited power of any single machine, distributed computing is an area of huge interest to those involved in high performance computing. Network transparency allows a properly-structured CSP application to run on one computer or on many networked computers in parallel, with little or no change to the program itself. This advantage is particularly useful for problems that have a large variety of scale. For example, machine learning algorithms can operate on data sets that have a huge range of sizes – the smaller data sets can be learnt from using a single machine, but larger problems may require clusters of machines.

Networked computing is not just of interest with relation to high performance computing however. Any client/server application could benefit from the relationship being able to "stretch" over a network with little code change. For example, a GUI client with a server

backend could either run in a self contained application on one machine, or it could put the backend on another machine, perhaps shared between many clients.

One year ago, C++CSP was introduced at this conference [3], and at the beginning of this year was released under the Lesser GNU Public Licence [4]. It contained many useful new features such as templated channels and stateful poisoning (see section 2.1), but initially it had no built-in network support. During the intervening year network support has been integrated into C++CSP, and other new features have been added. This paper describes this work, the design decisions that were taken, justification for the decisions and the results that have been achieved with the network component. This paper assumes some familiarity with the basic concepts and API of the core C++CSP library.

2 New C++CSP Features

2.1 Non-poisonable Channel Ends

Stateful poisoning of channels is a feature of C++CSP whereby a process with an end of a channel can poison that channel, causing all successive attempts to use the channel in any form (apart from to poison it) by anyone to throw a `PoisonException`. However, allowing all processes with a channel end to poison the channel is not always desirable. For example, we may not want *anyone* with a writing end of an any-to-one channel to be able to poison it. To this end, channels also now have `noPoisonReader` and `noPoisonWriter` functions that provide channel ends that cannot be used to poison the channel (unlike the ends obtained from the `reader` and `writer` functions). Using this mechanism, processes can choose whether the channel ends they pass to their subprocesses can be poisoned or not.

2.2 Channel Factories

When two or more processes function by communicating mostly with each other, it is common to create a parent process that simply creates these two processes and provides an interface to them to the world, hiding the communications that exist solely between the child processes. For example, there might be a process that handles requests to access some data, and an interface process that provides this service to the world in a simple interface. These two processes could then be packaged up into a single server process. A tough decision is what sort of channel to have between the two internal processes. A one-to-one channel would be in keeping with the standard CSP semantics but would make other requests block until the data access process has finished with the request. A buffered channel may be desired instead; but there are multiple types of buffering – various sizes of FIFO, an overwriting buffer, or an infinite size FIFO.

The best solution would be to allow the channel policy to be specified. The channel could not be passed into the constructor as channels themselves cannot be copy-constructed. Pointers to channels could be passed but this constructor-parameter-per-channel approach would not scale well if the policy needed to be specified for a large (or variable) number of channels. Templating the process is possible but would not allow for the decision to be flexibly made at run-time. A buffer type could be passed to the process but this eliminates the possible speed-up from using unbuffered channels, and also does not allow for any future types of channels that may have significantly altered semantics (C++CSP has been designed to allow for such possible future additions). The solution chosen is to use channel “factories”.

A class factory (an OOP pattern) is an object that creates other objects. A channel factory is a class that can be used to create channels. Functionality is provided through inheritance,

and each channel it creates is guaranteed to have the same lifetime as the factory. This allows channels to last for a long time, but also can make them last beyond the lifetime of their ends. The syntax for the factory is as follows:

```
class SomeProcess : public CSProcess
{
    ChannelFactory<int>* factory;
protected:
    void run()
    {
        Chanin<int> readingEnd;
        Chanout<int> writingEnd_NotPoisonable;
        factory->one2One(&readingEnd,&writingEnd_NotPoisonable,
            true,false);
    }
    ...  pass in factory through the constructor
};
```

The `one2One` method of the channel factory takes the address of the two channel ends to assign to, and also two parameters (with a default value of `true`) to specify whether the reading and writing ends respectively should be poisonable (see the previous section). Channel factories were also implemented in JCSP.net [5, 6].

2.3 Channel Bundles

Channel bundles are a concept introduced by KRoC [7, 8] (as "channel-types"). A channel bundle is a collection of channel *ends*. Bundling channels allows for easy representation of an interface. For example a process might have two request channels of type `string`, and a reply channel for each – one of type `int` and one of type `vector<float>`. Therefore a channel bundle could be produced to represent this interface containing four items: `Chanin<string>`, `Chanout<int>`, `Chanin<string>`, `Chanout< vector<float> >`. On its own this would just be a simple struct-like data structure. Channel bundles are reversible though, so the reverse structure of `Chanout<string>`, `Chanin<int>`, `Chanout<string>`, `Chanin< vector<float> >` could be given to another process wanting to use the original process. In KRoC, the bundles are used for mobile channels as follows:

```
CHAN TYPE MYTYPE
  MOBILE RECORD
    CHAN INT in?:
    CHAN INT out!:
    CHAN BOOL extra!:
:

MYTYPE? myin:
MYTYPE! myout:

myin, myout := MOBILE MYTYPE
```

To do the same in C++CSP:

```
typedef
    Bundle< bundle::In<int>,bundle::Out<int>,bundle::Out<bool> >
    MyType;
```

```
One2OneChannel<int> a,b;
One2OneChannel<bool> c;

MyType::Normal myIn(a,b,c);
MyType::Reverse myOut(a,b,c);
```

The constructors of the two bundles can take the actual channels as arguments, and get the channel ends appropriately, to make their use simpler and more elegant.

3 Networked C++CSP – Motivation

Unlike other programming methodologies, such as procedural object-oriented programming, CSP is well-suited to utilising networks because it is naturally network-transparent. Providing that CSP semantics are preserved, CSP processes are able to operate over any transport mechanism, whether it is a CSP kernel on a single machine, or a network between two machines. This is because CSP processes are perfectly encapsulated – there is no shared data between processes. The only way processes can communicate with each other is through recognised CSP primitives such as channels and also barriers and buckets. Barriers and buckets can be built using channels, so for the purposes of this paper only channels will be considered.

C++CSP's University of Kent-based sister technologies, KRoC [7, 8, 9, 10] and JCSP [11, 12] already have networked versions, KRoC.net [13, 14] and JCSP.net [5, 6] respectively. They provide both a proof-of-concept for network-enabling CSP technologies, and also a useful design reference. This allows C++CSP to continue its trend of learning from KRoC and JCSP while also trying to enhance them where possible and appropriate.

4 Networked C++CSP – Aims

Before design work can be undertaken on the networked capability of C++CSP , the aims of the project need to be clarified, to inform our decision-making.

The central goal is to enhance C++CSP by allowing two C++CSP machines[1] to communicate with each other using channels via a network. The primary consideration is utilising TCP/IP sockets for the underlying transport mechanism, as TCP/IP sockets are virtually ubiquitous in modern networking.

The network channels should be easy to use, and should be interchangeable with other channels as much as possible. That is, they should be useable in parallel communications, alternatives, extended rendezvous[15] and so on wherever possible. They should be poisonable, as other channels are. All these aims facilitate network-transparency, the aim being that a process communicating over a network channel should notice no difference from communicating over a standard channel.

It would be easy to state that the network-enabled library should be efficient. There are many measures of efficiency here however. Possible criteria include minimal network latency, maximum data throughput, and minimal loss of speed to the processes the machine is running locally. Which of these is more important than the others depends on the particular application so wherever possible the balance between them should be configurable.

[1] In this paper, a machine is defined to be a C++CSP program. Hence two C++CSP machines can run on the same physical machine.

5 Networked C++CSP – Design

The design of the network enhancements to C++CSP consists of many different areas. It is clear that a new component will be needed in the kernel to handle the new consideration of network traffic. This will be referred to as the NET system. Its job is to be the interface between the network and the C++CSP kernel. Its implementation is discussed in section 5.2. The network protocol for channel communications will also need to be decided on – this is described in section 5.3. Finally, consideration needs to be given to the Application Programming Interface (API) for network channels (section 5.1), and the associated issues to do with naming and finding channels (section 5.1.2).

5.1 The Network Class Structure and API

5.1.1 Channel Creation

To create a one-to-one channel and its associated ends the following code can be used:

```
One2OneChannel<int> channel;
Chanout<int> writingEnd = channel.writer();
Chanin<int> readingEnd = channel.reader();
```

This is clearly unsuited to being used for network channels. A network channel is not created entirely on one machine, it is inherently split between the two machines that it communicates between. Therefore allocating the channel as above is not the best solution.

Instead, each machine will allocate its own channel object, where the two channel objects correspond to one actual channel. So one end will have the following code:

```
Net2OneChannel<int> networkChannel;
Chanin<int> readingEnd = networkChannel.reader();
```

The other end will have the corresponding code:

```
One2NetChannel<int> networkChannel;
Chanout<int> writingEnd = networkChannel.writer();
```

These two channels will not be connected as they are above. To correctly connect the two channel objects, we need a shared identifier – a channel name.

5.1.2 Channel Names

Both KRoC.net and JCSP.net opted for using the concept of a “Channel Name Server” (CNS) in their network components. With this system every networked machine took the address of a CNS and queried it for the network address (e.g. IP address and port number) of the machine that corresponded to the name of a channel. This is a similar concept to that of the Domain Name Server (DNS) which is now ubiquitous on the internet.

C++CSP will probably also have this concept included in the near future, but currently there is only a standard explicit connection syntax. A channel name is simply of the form “[NetworkAddress[:PortNumber]/]ChannelName”. If the network address is not included, the channel is allocated on the local machine. This means that other machines that connect can request to be connected to this channel, but the channel will not immediately be connected to another machine. At this point use of the channel will block indefinitely until someone connects to it.

If the machine name is included, then the name is effectively a request to connect to a remote channel. The network address (either an IPv4 address or a DNS name) and port number uniquely identify a machine on the network, and the channel name is the channel to connect to on that remote machine. Channel names can contain any ASCII characters above 32, except the / character.

5.1.3 Accepter Channels

The methods described above allow for a single process to connect to a single remote network channel. This will be fine for many scenarios, but will not work for situations where one server process needs to be able to accept connections from a variable number of clients.

To solve this problem, C++CSP supplies *accepter* channels – a concept similar to the socket `accept()` call. The idea behind accepter channels is that they have a channel name as normal, but for each remote connection, instead of connecting it to the accepter channel, a new channel is created. This mechanism is invisible to the remote process making the connection – it does not know whether it connected to a normal channel or an accepter channel. A suitable syntax for this might have been:

```
Net2OneChannelAccepter<int> accepter;
Net2OneChannel<int> channel = accepter.accept();
```

The `accept()` call above would of course block until a remote connection was made to this machine. To better fit in with the rest of the library however it is best to make the accepter channel a proper channel (of channels!) as follows:

```
Net2OneAccepterChannel<int> accepter;
Net2OneChannel<int> channel;
accepter.reader() >> channel;
```

There is one remaining issue with this implementation though. In any communication between a client and a server there will inevitably need to be communication in both directions at some point. CSP channels are one-way however, so just by connecting as above, only one way communication would be possible. There are various possible methods for establishing the channel in the other direction.

A second accepter channel for the client to join would not work properly as it cannot be guaranteed that they would join in the same order to match the connections on the two accepter channels, i.e. it would be impossible to tell which connection on accepter channel B was from the same client as one on accepter channel A.

The first channel accepted could be used to send the names of other channels to connect to back to the client. So the client would receive an array of channel names to use, such as “server/input0”, “server/output1”, but this would be a redundant channel after the connections had all been made. Such an approach is valid, but for some purposes the problem can be solved more efficiently by providing the address of the client to the server when the connection is made, so that the server can connect back to the client. So for example the client connects to the channel “server/ping”, the server gets the client address and connects to “client/pong” (see Appendix A for an example of the use of accepter channels).

This mechanism is useful when only one client from a C++CSP machine will be connecting to the server, otherwise name conflicts prevent this mechanism being used. In future, work could be done in the area of connecting entire batches of channels in a single connection attempt.

5.1.4 Channel Transparency

When the original C++CSP library was developed, a decision was taken to have channel ends that provide a consistent API to the channels themselves. While this introduces an extra virtual function call into every channel action, it was considered worthwhile to be able to truly hide the channel type from the process using it. Every channel type uses the same channel end objects, so these ends are all that processes communicating over a channel see. It is up to the parent process to choose the channel type, and pass the channel ends to its child processes appropriately.

At the time of development of the original library this allowed for buffered channels to be used seamlessly in place of normal channels, but with the advent of the networked component of C++CSP, it also allows network channels to be used in place of normal channels.

5.1.5 Data Serialisation and Conversion

Sending data over a network is often not as simple as copying the bytes between two networked machines. A major problem in C++ is that variables are often stored as pointers, references, or complex data structures containing these items. Clearly a pointer is only valid on the machine that it is being used; sending it over the network makes no sense. Another consideration for some systems might be the endianness of the networked machines – if an integer value is to be sent between a little-endian machine and a big-endian machine, it will need to be converted.

The solution to these issues in the networked C++CSP API is to provide a default behaviour for transmitting items across the network (simply copying the bytes), but to allow (and indeed encourage and where possible require) the user to override this. A class, `NetConverter` is provided:

```
template <typename DATA_TYPE>
class NetConverter
{
public:
    std::pair<const void*,unsigned int> toNet(
        const DATA_TYPE* data);
    unsigned int fromNet(
        DATA_TYPE* data,std::pair<const void*,unsigned int> pr);
};
```

The `toNet` method converts local data (the `data` parameter) into raw bytes for sending over the network (the return pair of a pointer to the data, and its size). The `fromNet` method converts raw bytes (the `pr` parameter above) back into local data (the `data` parameter) and returns the amount of bytes it had taken from the buffer. The default converter simply uses `memcpy` to perform the conversion. This is suitable for "plain old data" types but not for others, so the user is encouraged to provide their own specialisation of the class to handle the conversion.

There are converters supplied for many STL containers (such as `string`, `vector`, `map`) but these are implemented in the safest manner; they convert each data object individually. This is the correct decision for, say, `vector< map< int,ComplexDataStructure > >` but for `vector<float>` where there may be millions of `floats` in the array, it is very inefficient. Therefore if the user is going to be sending such structures, they can (and should) provide a converter for these.

Unfortunately such optimisations cannot be determined by the library itself due to the nature of C++ (and its lack of reflection). The library would explicitly have to supply converters for all situations where a simpler converter would be sensible, and this is not feasible, so it is

left to the user of the library. Also, the optimisation may be unwise. For example to convert a `vector<int>` may require converting each `int` individually to change its endianness, so the decision is left to the user.

5.2 The NET System

There are a number of options to consider for how the NET system might interact with the network and the local C++CSP machine. It could use either blocking or non-blocking sockets. Non-blocking sockets would allow the system to avoid making a blocking call, a behaviour that has always been strictly avoided in C++CSP (since blocking would stop the execution of *all* C++CSP processes). However, the behaviour of non-blocking sockets varies a lot between different Unix[2]-clones [16], let alone between them and Windows[3]. This, combined with the messiness of asynchronous notifications that arise from using non-blocking sockets, means that in fact blocking sockets are the better option, due to their simplicity and portability. Blocking on these sockets can actually be avoided by using the `select()` call to check a socket's status before making any send or receive calls.

The NET code can either be in the same thread as the C++CSP machine, or in its own thread. Having it in its own thread seems like the best way to reduce the slow-down effects of checking the network on the main C++CSP processes. It also appeals to the accepted wisdom of having your network code in a separate thread (for example, [17]). This would also allow properly blocking calls to be made without worry of it slowing down the main system.

Initially, the NET system was implemented in a separate thread to the normal C++CSP system. However, it became apparent that communication between the two threads (and all the associated synchronisation) incurred almost as much slow-down as checking the network did. Therefore it seemed that checking the network in the same thread would minimise the network latency without making any difference to the slow-down in the normal system. This claim is explored in a benchmark later on in section 7.2.

5.3 The Protocol

The NET protocol is used between two C++CSP machines. It needs to support the full range of C++CSP (one-to-one, unbuffered) channel semantics. This includes normal communications, parallel communications, extended inputs, alternatives, and poisoning. It must also support making connections to channels (including the previously described accepter channels), and error conditions.

5.4 Semantics

The NET system ensures that it is invisible to processes at the two ends of a networked channel by preserving channel semantics. The NET system does not act like a buffer – it transparently forwards the data across the network, with no change to the synchronisation semantics of the application running on top of it.

5.4.1 Connections

The NET system will multiplex the channels that are connected between two machines over a single socket. No matter how many channels there are between two C++CSP machines,

[2]UNIX is a registered trademark of The Open Group

[3]Windows is a registered trademark of Microsoft Corporation

there will only be one actual socket connected between them. This saves the overheads of checking many sockets, but will not affect the data throughput of the channels.

5.4.2 Communication

The simplest protocol for channel communication would be one message per channel communication (from writer to reader). However treating the message being sent as the end of the communication for the writer would allow the writer to continue even if the reader had not read from the channel – a clear violation of CSP synchronised channel communication. The solution is for the reader to send an acknowledgement when the network channel has been read. This simple system allows for all the communication types described above:

- *normal communication*: the writer sends a message with the contents of the communication and then blocks/freezes. The NET system of the reading machine recieves the message and feeds it into the channel. The reader (who may have been waiting, or may arrive at the channel afterwards) then reads the message; the NET system then sends an acknowledgement, which unblocks the writer.

- *extended rendezvous*: the extended rendezvous is a communication where the writer sends a channel communication, and the reader receives it but does not complete the communication (i.e. free the blocked writer) until it has processed the information in some way. This is simple to achieve in the NET system, which simply does not send back its acknowledgement until the reader indicates that the rendezvous is complete.

- *alternatives*: `ALT`ing over a network channel involves placing a "hook" for notification for when a message arrives for the network channel, which can then be used by a guard in an `ALT`.

- *parallel communication*: performing communications in parallel in C++CSP consists of two phases. The first involves setting up all the communications that are to be performed and the second is suspending the process until all have been completed. So for a parallel output, the writer simply sends the message with the contents of the communication (as in the normal communication) during the setting up phase, and the communication is deemed complete when the acknowledgement is recieved from each reader. A parallel input works similarly, completing when all inputs have arrived.

5.4.3 Channel Identification

It has already been specified that channels will have a name that can be used to make the channel identifiable and unique (within a single machine). So one possibility for identifying the destination channel when a message is sent down the socket would be to provide the channel name. However, C++CSP does not currently impose a limit on the length of channel names, so descriptive names (which should be encouraged to prevent conflicts) would impose a performance penalty for communication. Instead, each end of a networked channel is given a 32-bit *channel ID*, unique within each machine.

5.4.4 Messages

Every message passed between NET systems will take the form of a triple; the channel ID, a size field, and then a stream of data (of that size, except for special messages). The channel ID will always be the ID on the machine to which the message is headed, regardless of the

stage of the communication. So a simple communication will put the channel ID on the *destination* machine in the ID field, the size of the data in the size field, and then the data. The receiving machine will then reply (when the communication has been completed) with a message containing the ID of the channel on the source machine, with a special value `NetAckType` in the size field, and no data.

Each machine maintains a lookup table containing the ID of each channel on the remote machine it is connected to. This way each machine knows which ID to send messages back to on the channel, removing the need to send both a destination and source channel ID with each message. This helps conserve bandwidth.

When a channel is poisoned, a notification is sent to the other end. A poison notification is similar to a reply message – it contains the channel ID (as always, for the message destination end), a special value `NetPoisonType` in the size field, and no data.

The other message types are the negotiations involved in connecting two ends of a network channel together. To connect to a remote channel, the connecting machine must send a message with the special value `RequestChannelConnect_NetChannelId` in the ID field, the size field will contain the ID of the channel on the connecting machine, and the data stream will be a null-terminated string containing the name of the channel being connected to. In reply there will be a message with the special value `ReplyChannelConnect_NetChannelId` in the ID field, the ID of the channel on the replying machine in the size field, and the same string in the data stream.

6 Threading

C++CSP originally used a many-to-many threading model. Each C++CSP program contained one or more threads, which each contained one or more processes. This allowed for considerable flexibility in the library, and the possibility of taking advantage of multi-processor machines. Threads of the same program have a shared address space. This has the advantage of being able to send pointers between threads (rather than sending an entire data structure), although this in turn has the disadvantage of potentially letting the programmer break the CREW (Concurrent-Read, Exclusive-Write) principle. Having multiple programs (communicating via the NET system) would also allow advantage to be taken of multi-processor machines; the only difference would be not having the problematic shared address space, and the speed difference.

There is another issue aside from the speed difference however. Inter-thread communications are done in C++CSP using semaphores. This has the advantage of being portable and generally quick, but there is no portable equivalent of the socket `select()` call for semaphores that allows multiple semaphores to be checked (and potentially blocked on). This means that communicating via the NET system will scale *much* better than communicating between threads via semaphores. Of course, communicating between threads via the NET system has the same speed as communicating between programs using the NET system.

Therefore to keep the API as simple and clean as possible, C++CSP has been changed back to a single thread for processes to run in, and the advice for those wanting OS-level concurrency on a single physical machine is to run multiple C++CSP machines (programs) on the same physical machine.

6.1 Threading for the NET System

One option for adding the networking would be to run it in a separate thread to our single main thread. However the communication and scheduling overheads for communicating with

this thread cause problems. The network process would not (on most operating systems) be able to block while (doing the equivalent of) `ALT`ing between socket events and semaphore events. The only viable thing to do would be to use sockets to communicate between the individual threads and the network thread, which would then communicate using sockets to the remote machines. Clearly this middle-man will only slow the process down for no gain.

Including the network code in the main thread decreases network latency but will also increase the latency of local processes, because of the time taken to check the network. There is a direct trade-off between network latency and local latency, and as such it will be configurable. The benchmarks measuring network latency and local overheads can be found in sections 7.1 and 7.2 respectively.

7 Benchmarks

Previous sections have discussed the API of the networked part of the C++CSP library. The other important factor in distributed computing is the speed of the communications over the network. The following benchmarks were all carried out with a Gentoo GNU/Linux desktop machine with an Athlon 1600+ XP processor and 256MB of SDR RAM as the client (or single machine in single machine benchmarks), and where a second machine was needed as a server, a Windows XP machine with an Athlon 2400+ XP and 512MB of DDR RAM. The machines are connected via a 100Mbps LAN, which had no other load during the tests.

7.1 Latency

7.1.1 Explanation

This benchmark tests the delay involved in communicating between two machines. The easiest way to measure latency is of course a ping test. The time taken to send a (minimal) communication from A to B, and a reply from B to A, is timed by A to form a *ping* time. Usually machines are pinged using ICMP/IP, a protocol designed for such things. ICMP is usually responded to by the system kernel, whereas pinging between two processes over TCP/IP involves extra latency because the message has to pass from the kernel to the user process (which has to be scheduled by the operating system).

7.1.2 Results

A ping test was done between the two test machines, and over 50 runs the average ping time was 718 microseconds. As a comparison, a standard ICMP ping between the machines yielded an average time of 321 microseconds over 50 pings. This is mostly accounted by the extra acknowledgements over the network generated by the C++CSP NET infrastructure (to maintain CSP channel synchronisation), which in this case doubles the network traffic.

A slightly cleaned-up version of the code is provided in Appendix A, as it is a good small example of how to use the networked part of C++CSP.

7.2 Network System Overheads

7.2.1 Explanation

Network system overheads are the amount of time that the network system of C++CSP spends checking the network, i.e. the amount of time “lost” by the local processes because of having the network component running at the same time. The benchmark that will test this runs the

CommsTime benchmark [18] locally, while running a ping client (see the previous benchmark). That client is pinging another machine but, more significantly, is checking the network periodically. These times can then be compared with those from running *CommsTime* without starting up the network component of C++CSP – see Table 1.

7.2.2 Network Overheads Results

Table 1: Results from the Overheads Test

Network-check Frequency	Time Per CommsTime Loop (microseconds)
1 millisecond	2.754
500 milliseconds	2.814
No network	2.753

These timings are of absolute time, not CPU time. The results show that the networked component of C++CSP does not slow down the local processes, even checking a thousand times a second. This is a useful result, because it shows that supporting the network does not "cost" the local processes any significant time.

7.3 Communication Overheads

7.3.1 Explanation

This class of benchmarks measures the time taken up by the communications involved in distributing the computing across multiple machines on the network. This was measured by performing this benchmark with both the server and client on the same machine using normal C++CSP channels, and then running the server and client on separate machines using network channels. So communication overheads encompasses latency and network system overheads, in a realistic example of distributing a computation problem using C++CSP.

The benchmark chosen for this is calculating fourier coefficients. This is done by using the trapezium method of integration. It is an easily parallelisable task, as each calculation of a coefficient is independent of the others. It can therefore show how much difference the network communications are making to the speed of the benchmark.

Benchmarks were run to calculate 10,000 coefficients, splitting them up into work packages (WPs) of 1, 10, 100, 1,000, or 10,000 coefficients. Each coefficient was calculated using either 10, 100 or 1,000 trapeziums – giving a total of fifteen different benchmarks. Each work package requires two communications – one to send the details of the work (40 bytes) to the client, and one from the client to the server with the results (4**c* bytes, where *c* is the number of coefficients calculated in that work package).

These benchmarks were run on the same machine (using normal non-network channels), and then with the server on a different machine across the (100 Mbps) network, so that the client (doing the calculations) is on the same machine for all the tests. The results for all tests were averaged over 50 runs, to eliminate any anomalies from any other background tasks on the machine "stealing" some processor time during the timed tests.

7.3.2 Results – Server and Client on the Same Machine

Table 2 shows the results from running the benchmark on the same machine. The results show that no matter how the work is split, the work scales linearly with the amount of calculation needed.

Table 2: Times from the Same Machine Test (seconds)

Trapeziums	10^0 WPs	10^1 WPs	10^2 WPs	10^3 WPs	10^4 WPs
10	0.085583	0.084918	0.084831	0.086672	0.101007
100	0.818286	0.816175	0.815787	0.817667	0.831389
1000	8.101215	8.098428	8.094989	8.145628	8.105675

This means that the amount of communications that take place make no significant difference to the performance of the system for local channels – an encouraging result for non-networked C++CSP. The one result that deviates slightly from the strictly linear pattern is the test with the greatest number of work packages (only one calculation per work package) and the least calculation per work package (only 10 trapeziums). It makes sense that the communication overheads are mostly likely to show on the test with the smallest computation to communication ratio. Even for this extreme fine granularity, however, the effect is not marked – about 18%.

7.3.3 Results – Server and Client on Different Networked Machines

Table 3 shows the results from this test. It is clear that over the network the results scale both with the number of work packages and the trapeziums. The test with one work package has virtually no network overheads because only two communications are involved. This can be checked by comparing the network result for 1000 trapeziums (i.e. as the problem scales up) with the non-network result – there is only a 0.5% difference. So the one work package result can be used as a base case against which to compare the other networked results (in order to see what difference the greater amount of network communications is making).

Table 3: Times from the Networked Test (seconds)

Trapeziums	10^0 WPs	10^1 WPs	10^2 WPs	10^3 WPs	10^4 WPs
10	0.102680	0.104186	*0.158208*	*0.413571*	*4.108936*
100	0.836837	0.840752	0.877223	*1.231270*	*4.110065*
1000	8.139110	8.148298	8.195977	8.602985	*12.305083*

For higher numbers of trapeziums (i.e. for higher computation to communication ratios), the results are constant for 1, 10 and 100 work packages – so the overheads make no difference for these numbers of work packages (which have a linear relationship with the amount of communication). As the number of work packages climbs towards 10,000, the time taken starts to increase. 10,000 work packages means two network communications for each computation of a coefficient – a very low computation to communication ratio. By comparing the results for 10,000 work packages with those for one work package, it is fairly clear that the communication overheads for 10,000 work packages are around 3 to 4 seconds. Each work package involves four network communications (one each for the request and reply, plus one for each acknowledgement generated by the NET infrastructure), so this overhead is just under 100 microseconds per communication.

It should be noted that these benchmarks are not designed to show application *speed-up* – only network overheads. Farming out the client work (i.e. *all* the computational work) to more than one machine would demonstrate that. Table 3 indicates that near linear speed-up should be obtained for all combinations of numbers of trapeziums and work packages *except* for those whose timings are italicised.

8 Availability

The latest version of the C++CSP library, including all the development covered in this paper, is available from the website at `http://cppcsp.net/`, under the *Lesser GNU Public Licence*.

The latency tests (section 7.1) can be found in the C++CSP distribution in the directory `moretest/`; the files are `ping_server.cpp` and `ping_client.cpp`. They can be run using the `./More_Test -y` and `./More_Test -z` commands for server and client respectively.

The network system overheads tests (section 7.2) are included in the `ping_client.cpp` file in the `moretest/` directory and the `commstime_test.cpp` file in the `simpletest/` directory, both in the C++CSP distribution. The tests can be run using the `./More_Test -x` (using `./More_Test -y` for the ping server) and `./Test -j` commands.

The communication overhead tests (section 7.3) are located in the `fourier_server.cpp` and `fourier_client.cpp` files in the `moretest/` directory in the C++CSP distribution. They can be run using `./More_Test -f` and `./More_Test -g` for the server and client respectively.

9 Conclusion

Networked channels have been successfully integrated into C++CSP. The API is similar to the existing C++CSP API and the new network channels are interchangeable with existing channels. Thus, processes do not need to know whether the channel ends they are using belong to normal channels or network channels. This makes it very easy to switch between networked and non-networked versions of C++CSP applications. As home networking and the Internet grow ever more popular, this greatly enhances the usefulness of the C++CSP library.

The *ping* test gave a ping time between two networked processes as double that of an ICMP ping, which itself could be viewed as the maximum possible performance for a ping. This result is still under a millisecond, which can be considered to be an acceptable result. The network overhead test demonstrated that checking the network does not slow down the local processes, which helps remove doubts concerning the possible performance penalty from having the code checking the network in the same thread as the local processes. The last benchmark showed that if the work is divided up into suitably-sized work packages, the performance of distributed computing using networked C++CSP can scale almost linearly.

The (TCP/IP) sockets API provides untyped asynchronous reliable communication between networked machines. The C++CSP network system provides a typed synchronised (reliable and multiplexed) layer on top of the sockets API that abides by CSP semantic rules for channel communication and choice (represented by `ALT`ing). It is hoped that this networked C++CSP may provide a useful API and support infrastructure for applications that just want simple networking, as well as for those wanting networked CSP.

One planned addition to the library is the ability to communicate over “raw” TCP/IP sockets. C++CSP network channels are the preferred method of network communication as they observe CSP channel semantics and are typed, but C++CSP programs may need to communicate using an existing protocol on a raw TCP/IP socket (for example, HTTP). Allowing C++CSP programs to use sockets in the same way as channels would enhance the library’s usefulness in creating networked applications.

For the future, the provision of *Channel Name Services* (as in KRoC.net and JCSP.net) for a more flexible means of establishing network connections dynamically will be considered. There is also the question of sending channel ends or processes over the network, as discussed in [19, 14].

References

[1] C.A.R.Hoare. Communicating Sequential Processes. In *CACM*, volume 21, pages 666–677, August 1978.

[2] C.A.R.Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[3] N.C.C. Brown and P.H. Welch. An Introduction to the Kent C++CSP Library. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.

[4] GNU. Lesser GNU Public Licence. `http://www.gnu.org/licenses/lgpl.html`.

[5] Quickstone Technologies Ltd. JCSP Network Edition Home Page. Available at: `http://www.quickstone.com/xcsp/jcspnetworkedition/` Retrieved July, 2004.

[6] P.H.Welch, J.R.Aldous, and J.Foster. CSP networking for java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002.

[7] P.H. Welch and F.R.M.Barnes. Kent Retargetable occam Compiler Home Page. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/` Retrieved July, 2004.

[8] P.H.Welch and D.C.Wood. The Kent Retargetable occam Compiler. In *Proceedings of WoTUG 19*, volume 47, pages 143–166, March 1996.

[9] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Parts I and II. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–380, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[10] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2):121–136, April 2003.

[11] P.H. Welch. Java Communicating Sequential Processes Home Page. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/` Retrieved July, 2004.

[12] P.H.Welch. Process Oriented Design for Java: Concurrency for All. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, volume 1, pages 51–57. CSREA, CSREA Press, June 2000.

[13] Mario Schweigler, Fred M.R. Barnes, and Peter H. Welch. Flexible, Transparent and Dynamic occam Networking With KRoC.net. In J.F.Broenink, editor, *Communicating Process Architectures – 2003*, volume 61 of *Concurrent Systems Engineering*, pages 107–126, Amsterdam, The Netherlands, September 2003. (WoTUG), IOS Press.

[14] M. Schweigler. Adding Mobility to Networked Channel-Types. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575, pages 201–218, IOS Press, Amsterdam, The Netherlands, September 2004.

[15] F.R.M.Barnes and P.H.Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings-Software*, 150(2):121–136, April 2003.

[16] Geoffrey Lee. Non-blocking sockets. `http://www.wychk.org/~glee/non-blocking.html`.

[17] Sun Microsystems. How To Use Threads. `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`.

[18] Roger M.A. Peel. A Reconfigurable Host Interconnection Scheme for Occam-Based Field Programmable Gate Arrays. In Alan G. Chalmers, Henk Muller, and Majid Mirmehdi, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 179–192, IOS Press, Amsterdam, The Netherlands, September 2001. IOS Press.

[19] F.R.M. Barnes and P.H. Welch. Communicating Mobile Processes. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, WoTUG-27, Concurrent Systems Engineering, ISSN 1383-7575, pages 201–218, IOS Press, Amsterdam, The Netherlands, September 2004.

Appendix A: Ping Example Code

```
class PingClientTester : public CSProcess {     // client
protected:
    void run() {
        ...  read in server address into the string connectTo
        Net2OneChannel<std::string> chanIn("pong");
        One2NetChannel<std::string> chanOut(connectTo + "/ping");

        while (true) {
            chanOut.writer() << ourName;
            chanIn.reader() >> theirName;
            sleepFor(Seconds(1));               // 1 second delay between pings
        }
    }
};

class PingHandler : public CSProcess {          // server
public:
    Net2OneChannel<std::string> n2o;
    One2NetChannel<std::string> o2n;
protected:
    void run() {
        Chanout<std::string> out(o2n.writer());
        Chanin<std::string> in(n2o.reader());

        try {
            while (true) {
                in >> theirName;
                out << ourName;
            }
        }
        catch (PoisonException e) {}
        in.poison();  out.poison();
    }
};

class PingServerTester : public CSProcess {
protected:
    void run() {
        Barrier barrier(1);
        Net2OneAccepterChannel<std::string> accept("ping");

        while (true) {
            PingHandler* ph = new PingHandler(mn);
            std::pair<Net2OneChannel<std::string>*,std::string> pr(&ph->n2o,"");

            accept.reader() >> pr;

            try {
                ph->o2n.connect(pr.second + "/pong");
                spawnProcess(ph,&barrier);      // only spawn if there is no poison
            }
            catch (PoisonException) {
                ph->n2o.reader().poison();
            }
        }
    }
};
```