

# THE DESIGN OF A FREE CSN API MODEL

GAIUS MULLEY

*Department of Computer Science  
University of Glamorgan  
CF37 1DL  
E-Mail: gaius@gnu.org*

**Abstract:** This paper presents a design of a free CSN message passing model. A new free CSN implementation has been produced and preliminary performance results are also discussed together with the design implications.

*Keywords:* Modula-2, interrupts, pthreads, simulation, CSN.

## INTRODUCTION

The CSN (Computing Surface Network) was a message passing library designed and implemented by Meiko Scientific for use on the CS1 and CS2 series of supercomputers which it manufactured in the early 1990s. The CSN library is interesting as it is potentially lightweight and requiring few operating system services. The Meiko CSN library was implemented both on UNIX workstations and extremely lean micro-kernel systems. The library interface consists of nine core functions which provided non blocking and blocking communication mechanisms. The CS1 and CS2 series of supercomputers employed a MIMD architecture from between 2 and 1024 compute nodes. The compute nodes varied in specification and ranged from T400 [Inmos, 1984], T800 [Inmos, 1988a; Inmos, 1988b], i860 [Intel, 1992] and SPARC [Sparc International, 1992] based systems. Typically a potentially parallel application was divided into  $m$  distinct components and  $n$  common components. Each component would be built separately and the programmer would assign the various binaries to the compute resources available via a build program or a specification file.

This paradigm is worth revisiting as commodity architecture now available has very similar characteristic's to that of the CS1 series in the early 1990s. For example the Cell processor used in the PlayStation 3 has a 64 bit processing element and eight 32 bit processing elements and overall the Cell can be viewed as having a MIMD architectural model with SIMD on the individual processing elements. Irrespective of the Cell architecture many computing departments have a

number of underutilised 64 bit and 32 bit compute nodes. Harnessing this compute capability has always been problematic and the availability of another message passing library offers flexibility and a migration path for legacy parallelised programs.

## RELATED WORK

The CSN is a message passing library which was designed to allow the various heterogeneous processors in a supercomputer to interchange data. It was designed and implemented at Meiko scientific in the late 1980s for use on their supercomputers. The CSN was similar to PVM.

Parallel Virtual Machine [Geist, 1994] was originally developed at the University of Tennessee, Oak Ridge National Laboratory and Emory University in 1989. PVM is a software package which enables a variety of heterogeneous computers running a various operating systems to exchange data. In effect it provided the infrastructure necessary to implement grid computing.

The CSN slightly predates PVM and provides a simpler communication API. PVM provides facilities for fault tolerance, the CSN did not. The CSN provides support for both blocking and non blocking transmits and receives whereas PVM allows non blocking and blocking receives but only blocking transmits.

The simplicity of the CSN API together with the ability to handle non blocking transmits and receives make it attractive. It is well suited for use on a very fast, reliable and isolated local area network rather than in a grid computing scenario. Additionally the CSN is an attractive message passing interface to employ when implementing a parallel algorithm on a modern multi core machine.

## APPROACH

The approach to be taken in this work is to implement the CSN using a variety of programming languages. Python is to be used to parse the specification file, allocate processes to processors and initiate execution of a process on a remote processor. GNU Modula-2 [Mulley, 2006a] is to be used to implement the CSN library. GNU Modula-2 will utilise the portable GNU Pthreads [Engelschall, 2005] via its coroutine library [Mulley, 2006b]. GNU C will be used to interface between GNU Modula-2 and various library routines under GNU/Linux.

An important aspect of the work is that it should be free [Stallman, 2001] as defined by the Free Software Foundation. The obvious advantages are magnified in an academic environment where code can be studied, pulled apart, re-implemented and improved by students.

## OVERVIEW OF THE CSN

One of the key prerequisites to high performance is to overlap input/output with computation. An interprocess communication mechanism *must* be able to support non blocking network data transfer. For this reason the normal socket paradigm (client/server) is at a large performance disadvantage. The features of the CSN interprocess communication library which make it attractive when designing a high performance parallel system are:

- it is a transport level interface,
- it provides blocking transmits and receives,
- non blocking transmits and receives
- and multiple simultaneous non blocking receives and transmits.

The core CSN functions prototypes are presented in figure 1.

Initially an application must create a `Transport` via `Open`. Any process wishing to transmit data must firstly lookup the receivers transport `netid` using `LookupName` and conversely a program wishing to receive data must register a transport using `RegisterName`. A process may now transmit data either by call to `Tx` which will return once data has been received by another process. Alternatively it may call `TxNb` which starts transmitting data and immediately returns allowing the caller to overlap processing activity with the transmission. Each call to `TxNb` must be

accompanied by, at least, one call to `Test` which returns a status to indicate that this transmission has completed.

## SIMPLE CSN EXAMPLE

The procedure shown in figure 2 transmits a string `Hello world` to the sink procedure shown in figure 3. The `Source` procedure needs to lookup the `Sink` transport `NetId` before the transmission can take place. The transmission and reception in both examples are blocking. Internally within the CSN a blocking `Tx` and `Rx` is built from their non blocking counterparts followed by a blocking call to `Test`. The `LookupName` and `RegisterName` also serve to ensure that a transmit does not occur until the recipients transport has been initialised.

## CSN FILTER EXAMPLE

The filter example utilises a combination of blocking and non blocking transmits and receives. The filter application is broken into a three stage pipeline of processes, a source, filter and sink. The source and sink processes are very similar to the simple example, but operate with a larger buffer size.

The filter procedure is shown in figure 4 and figure 5. If the constant `MaxFilterBuffers` were set to two then the filter process effectively implements double buffering by initiating a non blocking receive at the same time as filtering the previous packet (`memset`). The performance results shown in table 1 were obtained when the `MaxFilterBuffers` was set to one, two and four.

## DESIGN

The free implementation of CSN uses GNU Pthreads which are non preemptive and it also uses GNU/Linux sockets which are blocking. It uses the GNU Modula-2 coroutine library [Mulley, 2006b] to ensure that before an input/output system call is invoked it will not block.

The implementation exploits inexpensive threads. A call to `Open` allocates a receiver thread on this `Transport`. This thread behaves as a server in traditional socket programming except that it creates a new thread (`rxWorkerThread`) for each incoming accept and these threads read data from a remote `TxNb`. Each new thread is associated with the `NetId` of the transport sending the data and each call to `RxNb` creates an incoming descriptor (`rxdesc`) which is placed onto the data structure identifying the transport, thread and incoming packet (`netidThread`). Once the receive is

```

PROCEDURE Open (VAR t: Transport) : CsnStatus ;

PROCEDURE Close (VAR t: Transport) : CsnStatus ;

PROCEDURE RegisterName (t: Transport; name: ARRAY OF CHAR) : CsnStatus ;

PROCEDURE LookupName (VAR n: NetId; name: ARRAY OF CHAR) : CsnStatus ;

PROCEDURE Tx (t: Transport; n: NetId; a: ADDRESS; l: CARDINAL) : CsnStatus ;

PROCEDURE Rx (t: Transport; VAR n: NetId; a: ADDRESS; l: CARDINAL;
              VAR ActualReceived: CARDINAL) : CsnStatus ;

PROCEDURE TxNb (t: Transport; n: NetId; a: ADDRESS; l: CARDINAL) : CsnStatus ;

PROCEDURE RxNb (t: Transport; a: ADDRESS; l: CARDINAL;
               VAR ActualReceived: CARDINAL) : CsnStatus ;

PROCEDURE Test (t: Transport; flags: CsnFlags; timeout: CARDINAL;
               VAR n: NetId; VAR a: ADDRESS; VAR s: CsnStatus) : CsnStatus ;

```

Figure 1: The core CSN prototypes.

```

PROCEDURE Source ;
VAR
  tpt : Transport ;
  sinkId: NetId ;
  buffer: ARRAY [0..12] OF CHAR ;
BEGIN
  Assert(csn.Open(tpt) = CsnOk) ;
  Assert(csn.LookupName (sinkId, 'sink') = CsnOk) ;
  StrCopy('Hello world', buffer) ;
  Assert(csn.Tx(tpt, sinkId, ADR(buffer), StrLen(buffer)+1) = CsnOk)
END Source ;

```

Figure 2: Procedure to transmit hello world

```

PROCEDURE Sink ;
VAR
  tpt : Transport ;
  actual: CARDINAL ;
  netid : NetId ;
  buffer: ARRAY [0..12] OF CHAR ;
  r : INTEGER ;
BEGIN
  Assert(csn.Open(tpt) = CsnOk) ;
  (* we register our transport *)
  Assert(csn.RegisterName(tpt, 'sink') = CsnOk) ;
  netid := NullNetId ;
  Assert(Rx(tpt, netid, ADR(buffer), HIGH(buffer)+1, actual) = CsnOk) ;
  r := printf('sink received string: %s\n', buffer)
END Sink ;

```

Figure 3: Procedure to receive hello world

complete the thread removes the descriptor from the `netidThread` and places it onto the `transports doneQ`. Thus when the receiving application calls `Test` the `doneQ` is searched and the appropriate values are returned. A blocking call to `Test` will either return immediately if the `doneQ` contained the

appropriate `rxdesc`, or it will result in the caller blocking on the transport until the `rxWorkerThread` releases it.

Conversely a call to `TxNb` results in the creation of a `txdesc` and it also checks to see that there exists a `txWorkerThread` which will transmit data from this

```

PROCEDURE Filter ;
VAR
    tpt      : Transport ;
    netid,
    sinkId   : NetId ;
    sink     : CARDINAL ;
    buffer   : ADDRESS ;
    r        : INTEGER ;
    i, actual: CARDINAL ;
BEGIN
    Assert(csn.Open(tpt) = CsnOk) ;
    Assert(csn.RegisterName (tpt, 'filter') = CsnOk) ;
    Assert(csn.LookupName (sinkId, 'sink') = CsnOk) ;
    FOR i := 0 TO MaxFilterBuffers DO
        ALLOCATE(buffer, BytesToBeSent) ;
        Assert(buffer#NIL) ;
        Assert(csn.RxNb(tpt, buffer, BytesToBeSent, actual) = CsnOk)
    END ;
END ;

```

Figure 4: Initialisation component of the filter process.

```

LOOP
    buffer := NIL ;
    netid  := NullNetId ;

    CASE csn.Test(tpt, {CsnRxReady, CsnTxReady}, MAX(CARDINAL),
                  netid, buffer, status)
    OF
        CsnTxReady:
            Assert(csn.RxNb(tpt, buffer, BytesToBeSent, actual) = CsnOk) |
        CsnRxReady:
            (* do some filtering on buffer - for now we set it to zero *)
            buffer := memset(buffer, 0, BytesToBeSent) ;
            (* now transmit it to the next process (a sink) *)
            Assert(csn.TxNb(tpt, sinkId, buffer, BytesToBeSent) = CsnOk) ;
    ELSE
        Halt(__FILE__, __LINE__, __FUNCTION__, 'csn.Test failed')
    END
END
END Filter ;

```

Figure 5: Main loop of the filter process.

transport to the remote NetId. The txdesc is then queued upon the appropriate netidThread. Once the transmission is complete the txWorkerThread moves the txdesc to the transports doneQ and releases any callers blocked on CSN Test. Figure 6 shows the relationship between these data structures during communication between a single receiver and transmitter.

## RESULTS

Table 1 shows the preliminary results obtained when running a three process filter pipeline on a Dual Core AMD Opteron™ processor 175 stepping 02 running at 2200 Mhz with 2 GByte of RAM. The dual core processor has 1 MByte of L2 cache, 64 KBytes L1

instruction cache and 64K data cache.

These performance figure are very preliminary as the implementation has just been completed. Nevertheless they are reasonably pleasing and show that the free CSN is a viable communication mechanism on modern multi core processors. It will be extremely interesting to find out the optimum packet size for highest throughput. A multiprocessor AMD Opteron [AMD, 2003] has a Hypertransport communication mechanism to allow memory transfer between processor cores. It will be interesting to see how much of the available 6.4 GByte/sec Opteron Hypertransport the Linux kernel and free CSN is able to utilise.

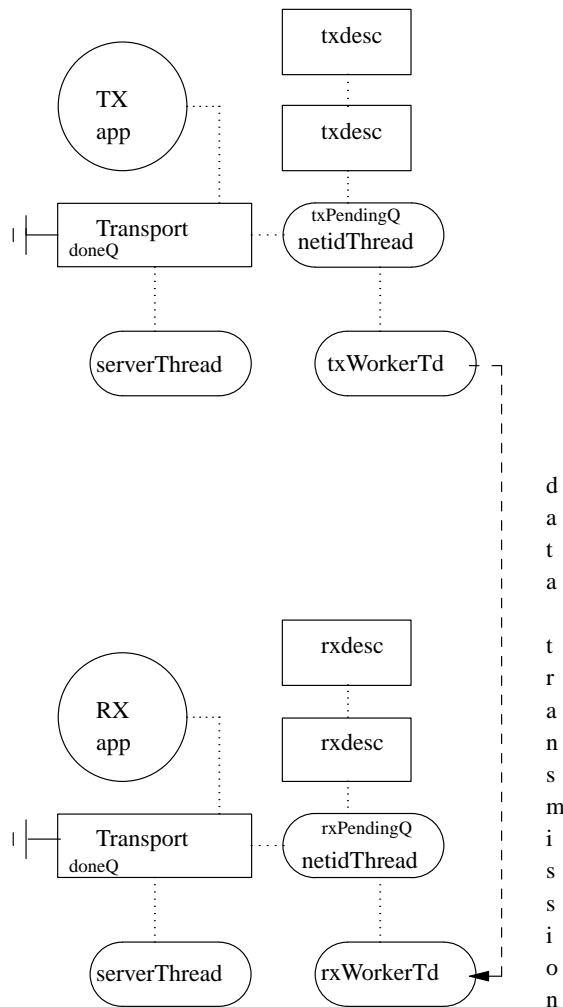


Figure 6: Relationship between the key data structure components during CSN communication

Block size (bytes)	Throughput (MByte/sec)		
	Single	Double	Quadruple
64	1	1	1
128	2	2	2
256	4	5	4
512	8	12	12
1024	14	11	23
2048	19	40	33
4096	47	37	70
8192	89	69	67
16384	137	137	178
32768	205	203	248
65536	283	274	306
131072	365	297	303
262144	329	285	257
524288	272	280	279
1048576	268	268	297

Table 1: Throughput of data in relation to buffersize

## CONCLUSIONS AND FURTHER WORK

In conclusion a free CSN message passing library has been implemented. The CSN API is considerably simpler than its more common PVM rival. The CSN is very well suited to modern multi core workstations and it provides non blocking transmits and receives.

Further work will be carried out investigating the performance of the CSN and completing the timeout mechanism within the `Test` routine. A python program to read a specification file and remotely allocate processes to processors will also be implemented. Finally the software will be packaged to conform with the standards of the GNU foundation.

## REFERENCES

- Inmos 1984, *IMS T414 Reference Manual*, Inmos Ltd, Prentice Hall.
- Inmos 1988, *Transputer Reference Manual*, Inmos Ltd, Prentice Hall.
- Inmos arch, "IMS T800 architecture," Technical Note 6, Inmos Ltd.
- Intel 1992, *i860 Microprocessor family programmer's reference manual*, Intel.
- Sparc International 1992, *The SPARC Architecture Manual*, Version 8, Prentice Hall.
- Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R. and Sunderam V. 1994, *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Scientific and Engineering Computation.
- Mulley G. 2006, *The GNU Modula-2 front end to GCC*, Edition 0.5, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
- Engelschall R.S. 2005, *GNU Pth - The GNU Portable Threads*, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
- Mulley G. 2006, "Cross Platform Simulation of the Interrupt Schema used within the GNU Modula-2 runtime system," UKSIM 2006, 9th International Conference on Computer Modelling and Simulation, Oriel College, Oxford, April 4-6.

Stallman R.M. 2001, *Using and Porting the GNU Compiler Collection*, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

AMD 2003, *Software Optimization Guide for the AMD Opteron Processor*, AMD, USA.