NML

The Neutral Messaging Language

NML Features

- NML is a software library for communication, ported to
  - Linux, Sun Solaris, SGI Irix
  - VxWorks, LynxOS, QNX
  - Microsoft Windows
  - Mac OS X
- NML applications running on one platform can communicate with ones running on any other platform
- Based on a message buffer model, with fixed maximum size, variable message length
  - Supports blocking and non-blocking reads
  - Supports queued and non-queued writes
  - Supports polled or publish/subscribe communication

NML Features (cont)

- Uniform application programming interface
  - same user source code regardless of computing platform
  - targeting new platforms requires recompiling, not recoding
- Protocol Independence
  - support for different protocols, e.g., shared memory, backplane global memory, TCP/IP sockets, UDP datagrams
  - new protocols can be added without affecting application code

NML Features (cont)

- Platform neutrality
  - conversion between chip data formats, e.g., big endian, little endian
  - independent of compiler structure padding
  - handles mutual exclusion
- Communication protocols specified in configuration file, not source code
  - locations of buffers and processes, selection of protocols and options are read by NML at run time
  - a running application can be extended dynamically

NML Availability

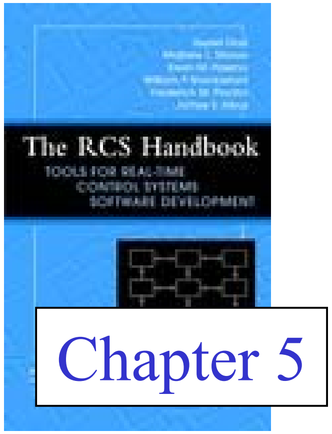
- NML is freely available
  - developed by U. S. Government employees as part of their official duties, and not subject to copyright
  - source code can be used for any purpose
    - understanding and debug
    - porting to new platforms
    - modification for commercialization
  - Links to all NML documentation at

`http://www.isd.mel.nist.gov/projects/rcslib/`

- See also The RCS Handbook, Veysel Gazi et al,

© 2001 John Wiley and Sons, ISBN 0-471-43565-1

NML Programming



Chapter 5

- C++ is the native language
  - messages are declared as C++ structures (classes)
  - C++ language bindings exists for all of NML
- Java is also supported
  - Java class code is automatically generated from C++ header files
  - Java language bindings exist for all of NML
- C “cover functions” can be written to give C applications access to NML
- Other languages can be supported using cover functions or via ASCII socket interface

Making Yours an NML Application

1. Define NML message vocabulary with C++ class declarations in header file
2. Use tools provided with NML to generate corresponding C++ or Java executable code
   - usually done in makefile
   - can be hand-written, if desired
3. Create NML configuration file specifying location of buffers and how processes connect to them
4. Create connections to NML buffers in your programs
5. Call read and write methods in your programs to transfer messages

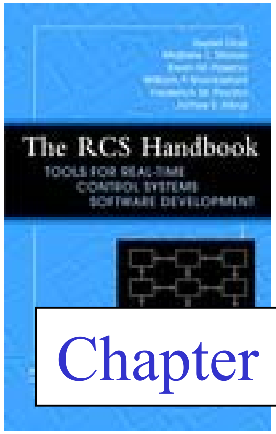
NML Messages

- Messages are declared as C++ classes derived from base class NMLmsg
- Programmers need to specify unique integer ID for each message
  - only needs to be unique among messages that may share a particular message buffer
  - typically unique within entire application, to ease development and debugging
- Programmers need to specify actual data fields in messages
- Message class also requires constructor and an update function
  - constructor initializes message and NMLmsg base class
  - update function is called by NML to read and write message
  - automatically generated
- Underlying protocol-specific work handled by Communication Management System CMS base class

NML Message Example

```
#include "nml.hh"

#define MY_MSG_TYPE 101

class MY_MSG: public NMLmsg
{
public:

  // constructor calls NMLmsg base class with type and size
  MY_MSG() : NMLmsg(MY_MSG_TYPE, sizeof(MY_MSG)) {};

  // update function is used by NML's communication management system
  // for reading and writing the message using builtin functions
  void update(CMS *);

  // user-defined data
  char c;
  double d;
};
/* Windows by default aligns doubles to 8-byte boundaries, Linux to
4-byte boundaries, so there are 7 bytes of padding after ‘c’ for
Windows, 3 for Linux. NML solves this ‘alignment problem’. */
```

Using the NML Code Generator



Chapter 10

- The Code Generator is written in Java, and reads and generates C++ code
- Saves tedium of writing required NML access functions for your messages
- Typically invoked via

```
java -jar <path>/CodeGenCmdLine.jar \
-I<path-to-other-headers> <yourheader.h>
```

- More complicated code generation can be accomplished by replacing `yourheader.h` with `script=<somescript.gen>`

NML Update Function Example

```
#include "rcs.hh"

void MY_MSG::update(CMS *cms)
{
  // call overloaded ‘update’ method for all your types
  cms->update(c); // CMS built-in update can handle chars
  cms->update(d); // CMS built-in update can handle floats, doubles
  // all other built-ins handled similarly
}
```

- Enumerations handled specially using `update_enumeration_with _name()` method
  - overcomes temporary copy problem
  - preserves symbol-to-value association, useful for diagnostics

NML Format Function

- All NML buffers contain the NML type ID, which can be read without any knowledge of the specific message
- NML uses the type ID to determine how to read or write the following message-specific data
- NML uses a format function for this
  - format function is provided to NML when processes create or connect to an NML buffer
  - the format function needs to be written by the programmer
  - it can be automatically generated, is using the RCS tool suite

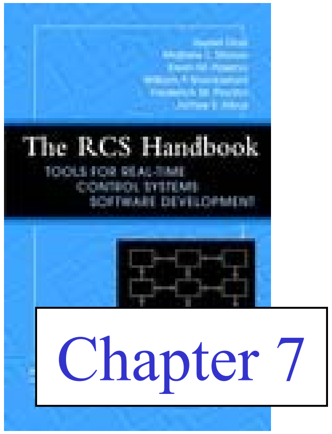
NML Format Function Example

```
#include "nml.hh"

int my_format(NMLTYPE type, void *buf, CMS *cms)
{
  switch(type) {
    case MY_MSG_TYPE:
    ((MY_MSG *) buf)->update(cms);
    break;

    /* others here */

    default:
    return -1;
  }

  return(0);
}
```

NML Message Vocabularies

- Message declarations and code declarations forms application vocabulary
  - Typically, all message declarations and format function declarations placed in a single header file
  - message definitions, format function placed in a single library
  - programming team references these
- Example: Enhanced Machine Controller (EMC) Vocabulary
  - EMC contains 131 messages for machine tool control
  - a single format function for all of these
  - one header file, emc.hh
  - one libary, libemc.a (or emc.lib for Microsoft platforms)
  - compiled for Sun Solaris, PC Linux, Windows NT

NML Configuration Files



Chapter 7

- NML uses text files to hold configuration information, not a central server or database
  - a single file may be stored on a central file server
  - many copies may be distributed redundantly
  - sections can be partitioned and distributed
  - all the usual pros and cons: inconsistent files v. server not there, etc.
- “New style” configuration files allow variable substitution, conditionals, file inclusion, default parameters, arbitrary parameter ordering, group defaults and other features
- Old-style config files are required at run time, and are generated from new style using `nmlcfg` (and vice-versa)
- more details follow

NML Buffers

- An NML buffer is a storage location with a fixed maximum size
- Messages of variable length can be written into NML buffers
- NML configuration file defines buffer parameters
  - ASCII text file
  - buffer lines begin with character B
  - comments begin with character #

Mandatory Buffer Parameters

- Name
  - a string used to identify buffers to processes who wish to create or connect to them
- Type
  - operating system shared memory, keyword SHMEM
    - for processes in a single operating system, mutual exclusion necessary
    - typical for Unix, Microsoft OSes
    - fast, with speeds typical of RAM access
  - backplane global memory, keyword GLOBMEM
    - for processes sharing common backplane, mutual exclusion necessary
    - typical for VxWorks in a VME bus or bus sharing, e.g., BIT-3
    - fast, with speeds typical of bus access

Mandatory Buffer Parameters (cont)

- Type (cont)
  - local (heap) memory, keyword LOCMEM
    - for a single process, no mutual exclusion necessary
    - provides NML API, anticipates multiple processes later
    - fast, with speeds typical of RAM access
    - can provide remote access if the single process is a server
      - becomes a database application
  - file (disk) memory, keyword FILEMEM
    - for processes sharing a file system, possibly networked
    - buffer is a disk file, useful for logging or scripting
    - slow, with speeds typical of file system latency
  - type-specific buffer parameters may follow at end

Mandatory Buffer Parameters (cont)

- Host
  - name of computer on which the buffer is located
  - used by remote processes to determine where to look for server
- Size
  - fixed maximum size, in bytes, to be allocated for the buffer
  - large enough for:
    - size of maximum message: your data plus 2 ints for type and size
    - 3 in-buffer int flags
    - 32 bytes for buffer name check
    - for GLOBMEM or mutex=mao_split (described later), one byte for each connecting process

Mandatory Buffer Parameters (cont)

- Neutral
  - flag signifying if data is to be written in native format or neutral format (e.g., XDR)
  - can be native (0) even if remote processes will connect, since NML converts to neutral for all network connections
  - may be neutral (1) to force data to be converted to neutral format in the buffer, if two different processor architectures share backplane global memory

Mandatory Buffer Parameters (cont)

- Buffer number
  - integer, generally different for each buffer
  - need only be different for buffers that share the same server
- Maximum processes
  - only relevant for GLOBMEM or mutex=mao_split
  - maximum processes that will connect to buffer
  - used to set aside in-buffer flags to enable mutual exclusion
  - not determined at compile time

Optional Buffer Parameters

- Remote access method
  - requires a server, described later
  - specifies which network protocol is to be used
  - either TCP/IP or UDP, with user-specified port number
  - simplified TCP/IP: conversion to text stream
    - allows easy interface for other languages
    - can interactively telnet into NML
  - keywords: TCP=<port>, UDP=<port>, STCP=<port>

Optional Buffer Parameters (cont)

- Alternative encoding methods
  - if neutral is selected, default is XDR
  - can also be ASCII, or display ASCII (better formatting for readability); keywords ascii, disp
  - XML, with optimization for sending changes only
- Confirm remote writes
  - for remote NML connections, specifies that write requests won't return until server sends acknowledge
  - keyword: confirm_write

Optional Buffer Parameters (cont)

- Queuing
  - messages can be queued in any buffer
  - size of buffer includes queue
  - keyword: queue
- Blocking reads
  - blocking reads block the calling process until a new message is received
  - requires an additional binary semaphore
  - keyword: bsem=<key>
- UDP read broadcasting
  - when server gets read request from anyone, or sends a subscription, it broadcasts to everyone
  - keyword: broadcast_port=<UDP port>

Type-Specific Buffer Parameters

- SHMEM operating system shared memory
  - key, same for both memory and semaphore; first entry after mandatory parameters
- GLOBMEM global (bus) memory
  - keyword vme_addr=<bus address>, e.g., vme_addr=0x400000
  - keyword vme_code=<which address space>, e.g., vme_code=0 (full address space), 1 (short address space)
- No additional for LOCMEM
- FILEMEM disk file buffers
  - key, for mutex semaphore required by some platforms; first entry after mandatory parameters
  - must specify neutral encoding, with disp keyword
  - default input file is standard input, output file is standard output
  - keywords: in=<script file>, out=<log file>
  - keyword max_out=<integer> sets size of output file; file is a ring buffer

NML Config File, Buffers (New Style)

```
# buff.nml2

# Shared memory, single operating system
b bufname=buff1 host=pc1 size=1024

# Global memory, single backplane
b bufname=buff2 host=vx2 size=280 buftype=GLOBMEM \
  vme_addr=0x40E00600

# Local memory, single process
b bufname=buff3 host=pc3 size=1024 buftype=LOCMEM
```

NML Config File, Buffers (Old Style)

```
# buff.nml
# NML file showing the four NML buffer types and sample
parameters

# Shared memory, single operating system

# Name  Type  Host Size Neut? (old) Buffer# MP Key
B buff1 SHMEM pc1  1024 0     0     1       4  1001

# Global memory, single backplane

# Name  Type    Host Size Neut? (old) Buffer# MP Bus Options
B buff2 GLOBMEM vx2  280  0     0     2       4  vme_addr=
       0x40E00600 vme_code=0
```

NML Configuration File, Buffers (cont)

```
# Local memory, single process

# Name  Type    Host Size Neut? (old) Buffer# MP
B buff3 LOCMEM  pc3  1024 0     0     1       4

# File memory, for logging or scripting

# Name  Type     Host Size Neut? (old) Buffer# MP file options
B buff4 FILEMEM  pc4  1024 1     0     1       4  in=script
       out=log max_out=1000
```

Mutual Exclusion for SHMEM

- Keyword: mutex=<type>
- os_sem, the default
  - use operating system semaphores
  - requires kernel call
- none
  - don’t do anything
  - fast, but will certainly cause problems unless application provides its own mutex method

Mutual Exclusion for SHMEM (cont)

- no_interrupts
  - disable interrupts to prevent multitasking
  - fast, but can compromise system peripherals
  - not all platforms support this; some require root privilege
- no_switching
  - like no_interrupts, but just disables tasking interrupt
  - peripheral interrupts are not affected
- mao_split (double buffering)
  - buffer is doubled, one part for reading, the other for writing
  - roles are interchanged after a write
  - very fast, since no kernel calls required
  - may lead to postponement, with occasional timeouts

Mutual Exclusion for GLOBMEM

- Via in-buffer process flags
  - Each process has an associated byte in the buffer, from the max process number from the buffer parameters
  - When reading or writing, process writes a 1 or 2, respectively, in its byte
  - Read operations poll for no other processes writing, then read and clear
  - Write operations poll for no other processes reading or writing, then write and clear
  - Timeout or indefinite postponement is a possibility
- Via bus locking
  - add keyword bus_lock to GLOBMEM **buffer line**
  - also requires **process line** keyword bd_type=<board type>, with MVME162 currently supported, MVME2700 a possible extension
  - bus will be locked during reads and writes
  - similar to mutex=no_interrupts for SHMEM

NML Processes

- An NML process creates or connects to one or more NML buffers
- Processes read and write messages
- More than one process may connect to an NML buffer, queued or non-queued
- NML configuration file defines process parameters
  - process lines begin with character P
  - one process line for each process-buffer connection

Mandatory Process Parameters

- Name
  - name of the process connecting to the buffer
  - set to “default” and this process line will be used by any process connecting to this buffer not specifically listed earlier
- Buffer
  - name of the buffer to connect to, matching a buffer line
  - set to “default” and this process line will be used by this process for any buffer for which no process line appeared earlier

Mandatory Process Parameters (cont)

- Type, keywords LOCAL or REMOTE
  - LOCAL means :
    - same host (SHMEM, LOCMEM)
    - same file system (FILEMEM)
    - same backplane (GLOBMEM)
  - REMOTE means not LOCAL
    - requires an NML server running LOCAL to the buffer
    - protocol for server specified at end of buffer line, e.g., TCP=5001
    - Java applications must use REMOTE
- AUTO means to LOCAL if the host on the buffer line matches the runtime hostname, REMOTE otherwise
  - may result ins some unnecessary REMOTE determinations since LOCAL access on the backplane may still be possible

Mandatory Process Parameters (cont)

- Host
  - name of machine on which this process is running
  - not currently necessary, but useful for documentation or possible future features
- Operations, keywords R for read, W for write, RW for read/write
  - not currently enforced: only a single warning is output
- Server flag
  - 0 if this process is not to be an NML server, 1 if it is, 2 if the process is not a server but should spawn a server
- Timeout
  - timeout, in seconds, for all NML operations

Mandatory Process Parameters (cont)

- Master
  - 0 if this process is not to create the NML buffer, 1 if it is
  - it's an error for a non-master process to connect to a non-existing buffer
  - it's not an error for a master process to connect to a non-existing buffer
    - multiple masters can be used to effect independent process start-up order
    - race conditions exist which must be addressed by application
- Connect number
  - a unique number between 0 and the buffer's max processes - 1
  - used as index into in-buffer process flags, for mutex

Optional Process Parameters

- bd_type=<board type>
  - used in conjunction with buffer option bus_lock
  - specifies the board type that will do the bus locking
  - MVME162 supported (bd_type=MVME162); others may be added (e.g., MVME2700)
- poll (detailed later)
  - only affects REMOTE process reads
  - reverses request for data from server with server send
  - reduces time spent in read operations, but doubles data latency
  - suitable for processes that read in synchronous bursts, e.g., GUIs

Optional Process Parameters (cont)

- sub=<secs> (detailed later)
  - only affects REMOTE process reads
  - server automatically sends to subscribing process at <secs> intervals
  - cuts out half of network traffic
  - suitable for processes that continually read synchronously
  - process must read as fast, or faster, than subscription rate, to prevent filling up network data queue
- UDP write broadcasting
  - keyword: broadcast_to_server=<net mask>
  - remote writes go to UDP port for entire subnet

NML Config File, Processes (New Style)

```
# proc.nml2

# applies to all processes
process_default timeout=1.0

# proc1 is the local master for buff1 and buff2, but must
access buff3 and buff4 remotely
p procname=proc1 bufname=buff1 proctype=local master=1
p procname=proc1 bufname=buff2 proctype=local master=1
p procname=proc1 bufname=buff3 proctype=remote
p procname=proc1 bufname=buff4 proctype=remote

# serv1 is the server for any buffer it connects to
p procname=serv1 bufname=default proctype=local server=1

# catch all in case we forgot anybody
p procname=default bufname=default proctype=auto
```

NML Config File, Processes (Old Style)

```
# proc.nml
# NML file showing NML process types and sample parameters

# Name  Buffer Type   Host  Ops Server? Timeout Master? Cnum

P proc1 buff1  LOCAL  comp1 R   0       1.0     1       0
P proc1 buff2  LOCAL  comp1 W   0       1.0     1       0
P proc1 buff3  REMOTE comp1 R   0       1.0     0       1
P proc1 buff4  REMOTE comp1 W   0       1.0     0       1

P proc2 buff1  REMOTE comp2 W   0       1.0     0       1
P proc2 buff2  REMOTE comp2 R   0       1.0     0       1
P proc2 buff3  LOCAL  comp2 W   0       1.0     1       0
P proc2 buff4  LOCAL  comp2 R   0       1.0     1       0

P serv1 buff1  LOCAL  comp1 RW  1       1.0     0       1
P serv1 buff2  LOCAL  comp1 RW  1       1.0     0       1
P serv2 buff3  LOCAL  comp2 RW  1       1.0     0       1
P serv2 buff4  LOCAL  comp2 RW  1       1.0     0       1
```

Opening NML Buffers

- A buffer is created by the master process(es)
  - master flag set to non-zero in NML config file
  - no error if already created (multiple masters possible)
- Other processes can connect to existing buffers
  - master flag set to zero in NML config file
  - error if not already created
- C++ example:

```
NML * buffer = new NML(formatFunction,
"buffer", "process", "config.nml");
```

Buffer Opening Details

- Buffer Verification
  - creating process writes a checksum into the buffer based on the buffer name
  - connecting processes compare the checksum, return an error if invalid
  - detects problems with duplicate keys in configuration file
- Copy-out allocation
  - each process that reads is allocated local heap memory into which read results are copied
  - processes that are write-only do not incur this allocation

Checking NML Buffers

- The “valid” NML method can be used to check result of buffer creation or connection
- NML error type contains reason for failure, if any
- C++ example:

```
if (! buffer->valid()) {
   printf("NML error: %d",
         buffer->error_type);
}
```

NML Error Types

NML_NO_ERROR	No error
NML_INVALID_CONFIGURATION	Format error in the configuration file
NML_BUFFER_NOT_READ	Write-if-read operation failed due to no prior read
NML_TIMED_OUT	The operation timed out
NML_FORMAT_ERROR	The update or format function failed, or the message is too large for the buffer
NML_NO_MASTER_ERROR	The process is not the master and the buffer ddoes not exist
NML_INTERNAL_CMS_ERROR	An operation failed in the low-level protocol handling

NML Write Operations

- All write operations take an NML message argument
  - either directly (as a reference), or via a pointer
- Overwrites any message in the buffer, unless the buffer is queued
- Returns 0 if successful, -1 otherwise
- Non-blocking write: write(NMLmsg *msg), write(NMLmsg &msg)
  - writes the message immediately
- Non-destructive write: write_if_read(NMLmsg *msg), write_if_read(NMLmsg &msg)
  - only writes the message if the buffer is empty
  - otherwise, does not write, and returns error

NML Write Example

```
#include "nml.hh"
#include "myvocab.hh" // user NML vocabulary declarations

// connect to NML buffer
NML * buffer = new NML(my_format, "command", "supervisor",
"myapp.nml");

// create an instance of the NML message to send
MY_MSG my_msg;

// fill it in
my_msg.f = 3.1416;
my_msg.c = 'P';
my_msg.i = 1;

// write it
if (0 != buffer->write(my_msg)) {
  printf("error writing: %d\n", buffer->error_type);
}
```

NML Read Operations

- All read operations return the NMLTYPE of the message in the buffer
  - 0 means no new message
  - -1 is an error
  - otherwise, it's a message
- The result of the read operation is pointed to by get_address()
- If a process reads a message from a non-queued buffer:
  - the buffer is empty for that process
  - other processes will still be able to read the message
- If a process reads a message from a queued buffer:
  - that message is removed for all processes
  - other processes will not be able to read the message

NML Read Operations (cont)

- Non-blocking read: read()
  - returns immediately
- Blocking read: blocking_read(double timeout)
  - blocks until a message has arrived, or until the timeout expires
  - NML configuration file needs bsem=<key> for the buffer
- Monitoring read: peek()
  - with respect to other processes, does not affect the buffer in any way, queued or non-queued
    - subsequently, other process write_if_read operations will fail
  - with respect to this process:
    - for a non-queued buffer, behaves just like a read
    - for a queued buffer, subsequent peeks will return no message until another read operation by any process

NML Read Example

```
#include "nml.hh"
#include "myvocab.hh" // user NML vocabulary

// ptr to MY_MSG class, which is to be read
MY_MSG *my_msg;

// connect to NML buffer
NML * buffer = new NML(my_format, "status", "supervisor", "myapp.nml");

// read it
switch (buffer->read()) {
  case 0: // no new message
  break;
  case -1: // NML error
  printf("error reading: %d\n", buffer->error_type);
  break;
  case MY_MSG_TYPE:
  my_msg_ptr = (MY_MSG *) buffer->get_address();
  printf("%f %c %d\n", my_msg->f, my_msg->c, my_msg->i);
  break;
}
```

Miscellaneous NML Operations

- Checking for an empty buffer: check_if_read()
  - peek operations do not affect this at all
  - for non-queued buffers:
    - returns 0 if the buffer contains a message that has not been read by any process
    - returns 1 if the buffer contains a message that has been read by at least one process
  - for queued buffers:
    - returns 0 if the queue has at least one message
    - returns 1 if the queue is empty
  - returns -1 if there's an error
- Clearing a buffer: clear()
  - clears all messages out of the buffer

Remote Access to NML Buffers

- Processes that can't connect locally to a buffer need help from a server
  - server runs local to buffer
  - server awaits requests from remote processes to read or write
- Server is a process, with server flag set in NML configuration file
- Server first creates or connects to buffers just like a normal process
- Server then calls special NML functions to initiate network services for opened buffers

NML Server Functions

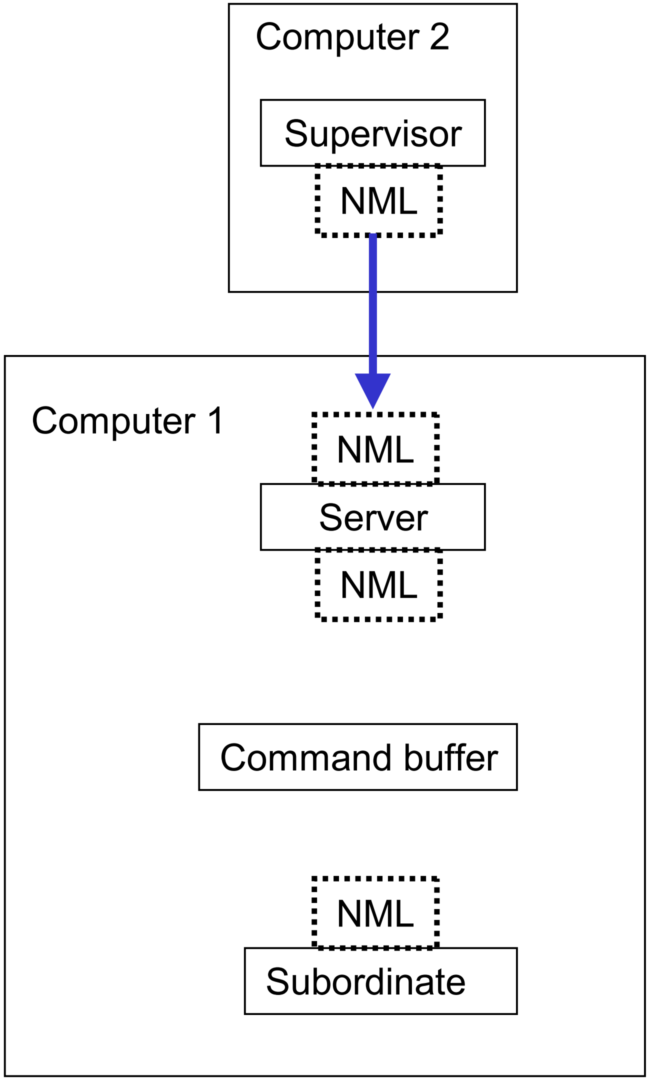
- run_nml_servers()
  - starts a server for each group of opened NML buffers that share a common network parameter
    - uses platform-specific tasking (e.g., fork/exec, taskSpawn) to initiate server task(s)
    - calling process made pending until wakeup signal
  - no return value
- nml_start()
  - like run_nml_servers(), except that it returns to calling process
- nml_cleanup()
  - stops the NML servers started by a call to nml_start, and closes the associated NML buffer connections

NML Server Example

```
#include "nml.hh"
#include "myvocab.hh" // user NML vocabulary

// this is it-- the whole program
int main()
{
  // connect to NML buffers
  NML * command_buffer = new NML(my_format, "command",
       "server", "myapp.nml");
  NML * status_buffer = new NML(my_format, "status",
       "server", "myapp.nml");

  // run NML server(s) for both buffers; doesn't return
  run_nml_servers();

  return 0;
}
```

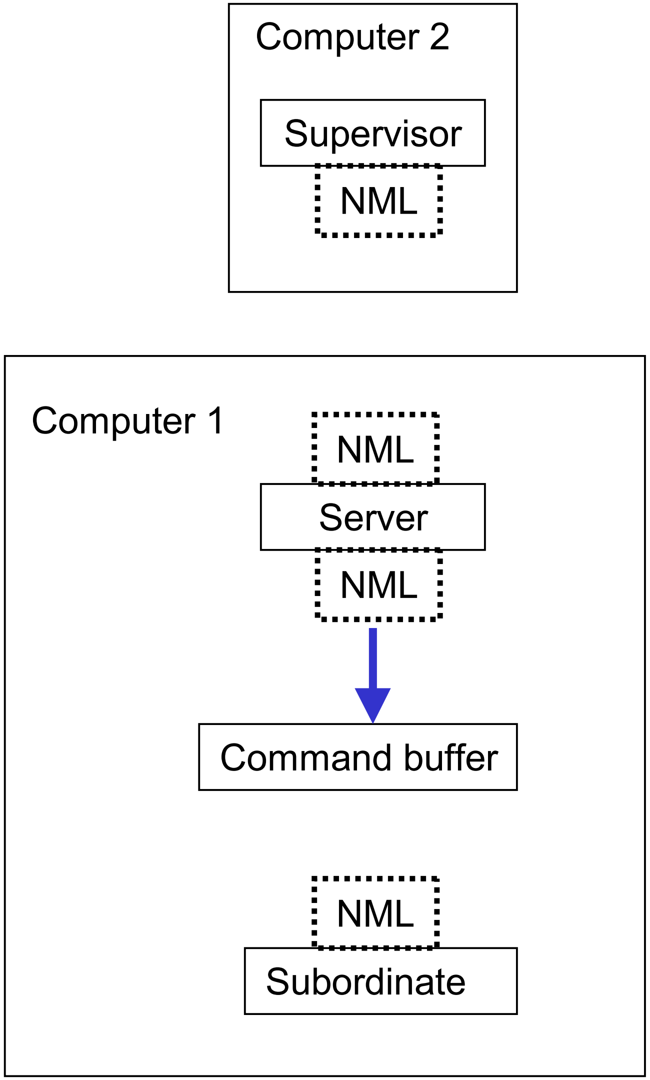
NML Server Example (cont)

```
#include "nml.hh"
#include "myvocab.hh" // user NML vocabulary

int main()
{
  // connect to NML buffers
  NML * command_buffer = new NML(my_format, "command",
"server", "myapp.nml");
  NML * status_buffer = new NML(my_format, "status",
"server", "myapp.nml");

  // start NML server(s) for both buffers
  nml_start();
  // run rest of application here
  // now stop NML servers(s)
  nml_cleanup();
  return 0;
}
```



Remote Write Operation

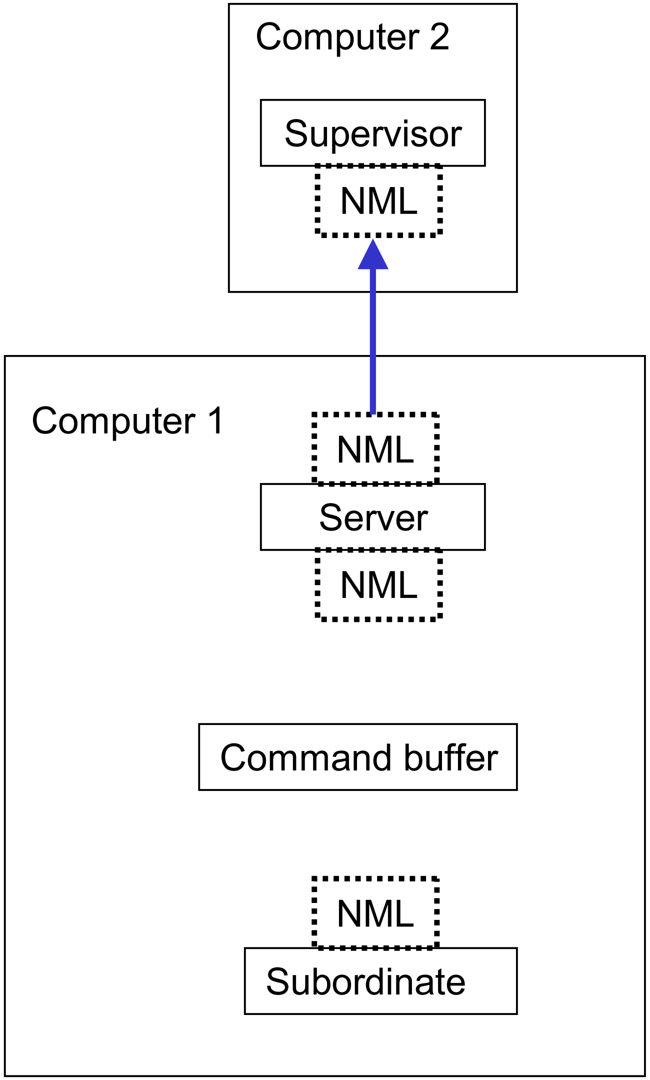
The remote writer (supervisor) sends a write request, with the message and the target buffer (command buffer), to the server. This is done by NML automatically. The application code is the same as for a local write; the NML configuration file is different.



Remote Write Operation

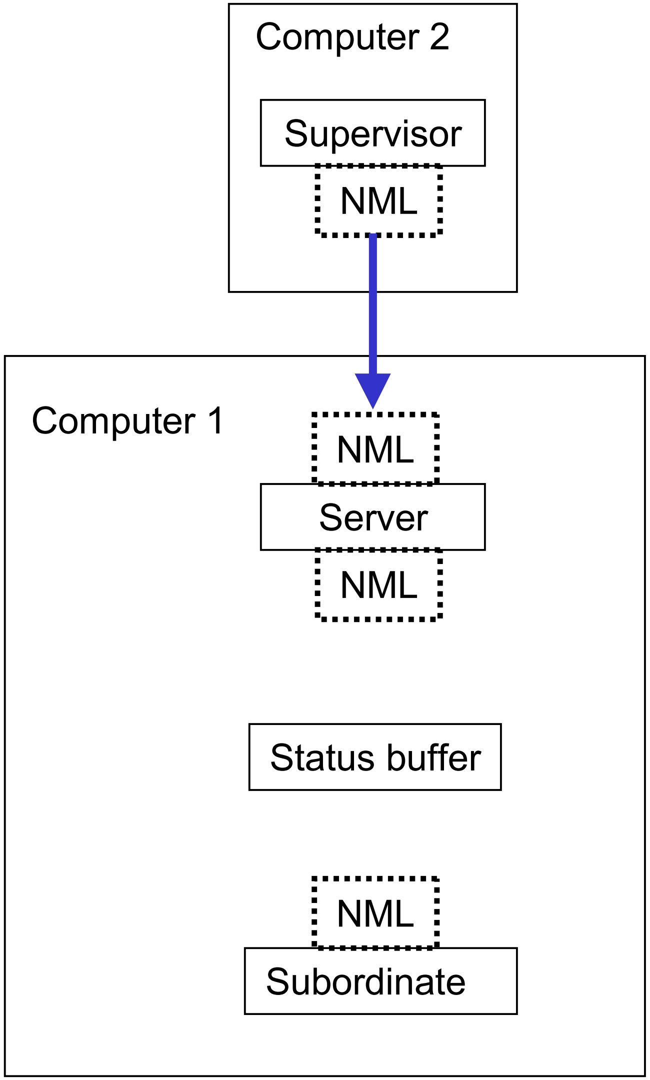
The server writes the message into the target buffer (command buffer), locally.

If the buffer parameters did not include confirm_write, the supervisor's write operation will return immediately.



Remote Write Operation

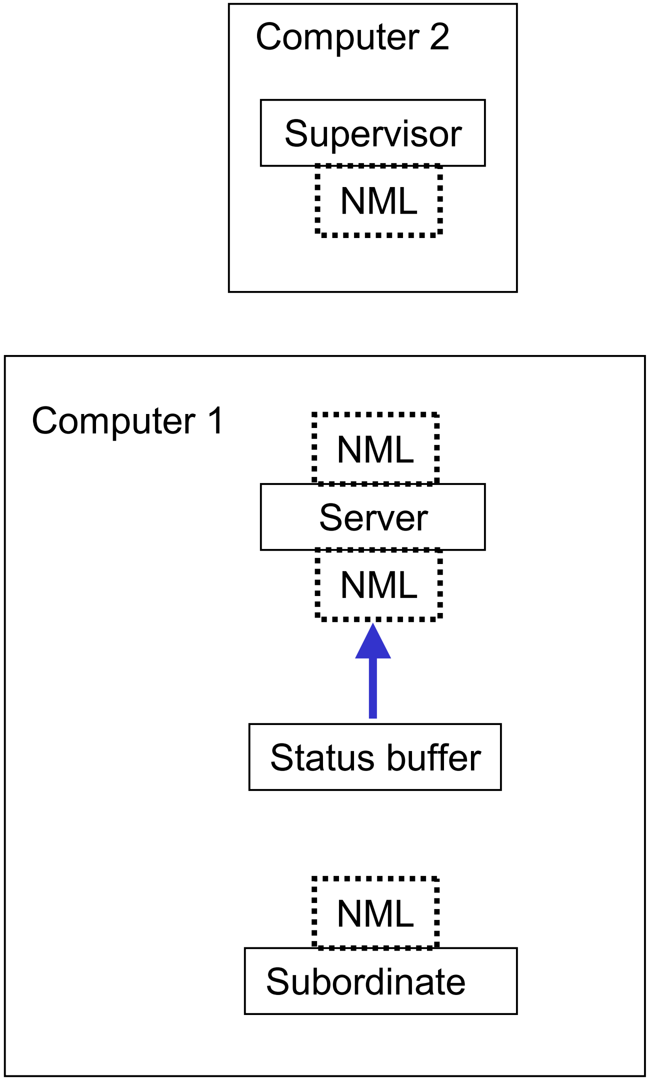
If the buffer parameters included confirm_write, the supervisor's write operation will wait until an acknowledge from the server.



Remote Read Operation

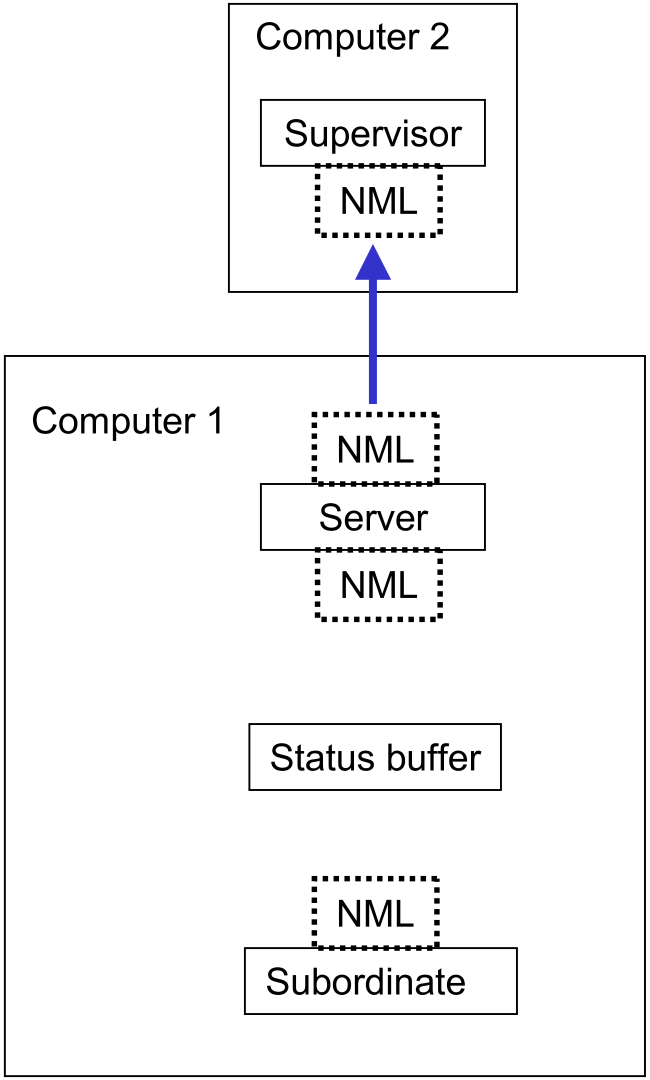
The remote reader (supervisor) sends a read request for the target buffer (status buffer) to the server.

This is done by NML automatically. The application code is the same as for a local write; the NML configuration file is different.



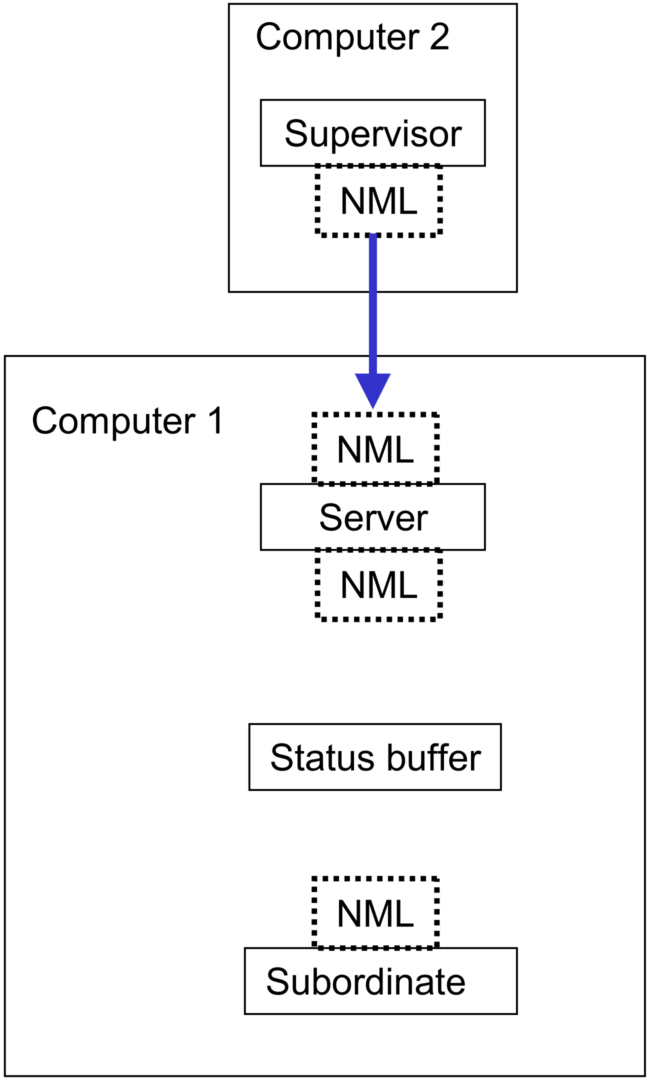
Remote Read Operation

The server reads the contents of the target buffer (status buffer), locally.



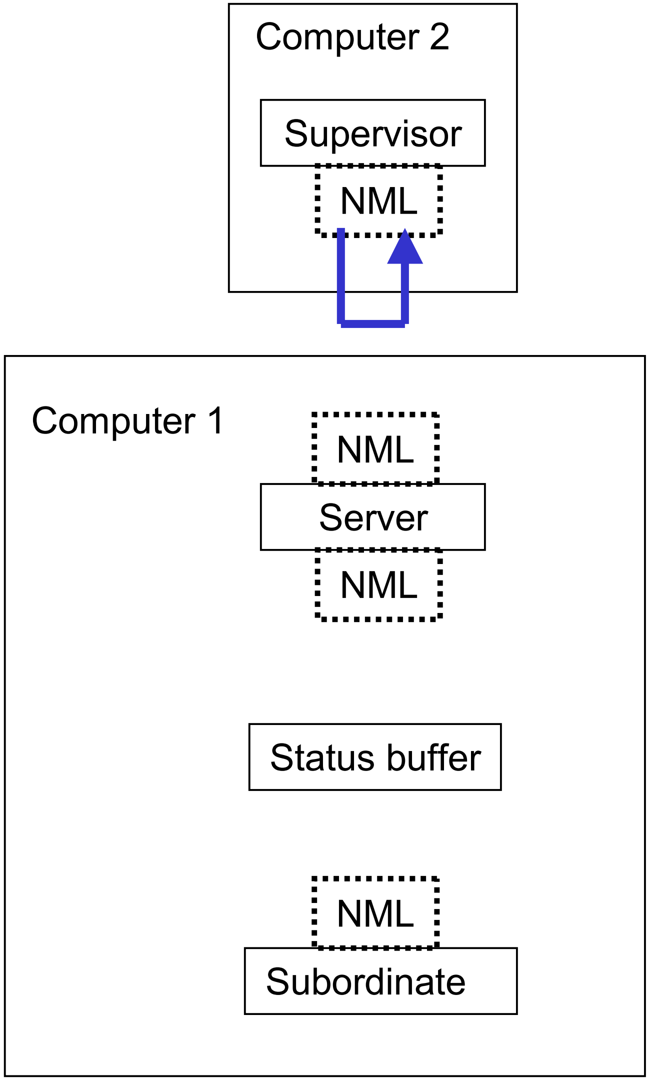
Remote Read Operation

Finally, the server sends the result over the network to the supervisor, whose read operation returns.



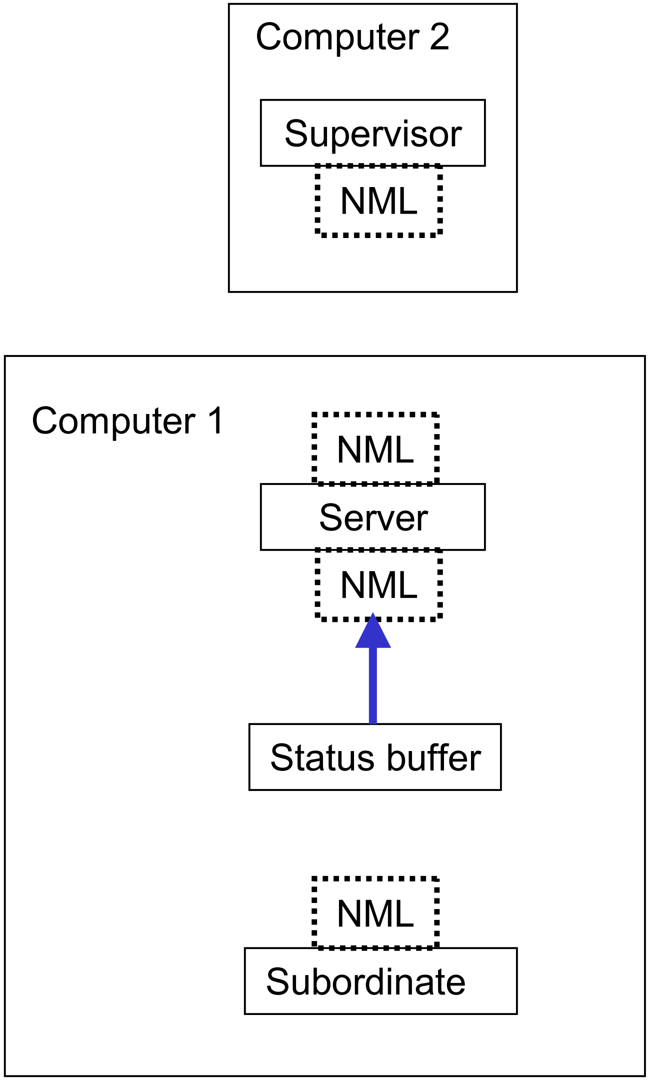
Remote Read Operation, Polling

The remote reader (supervisor) sends a read request for the target buffer (status buffer) to the server.



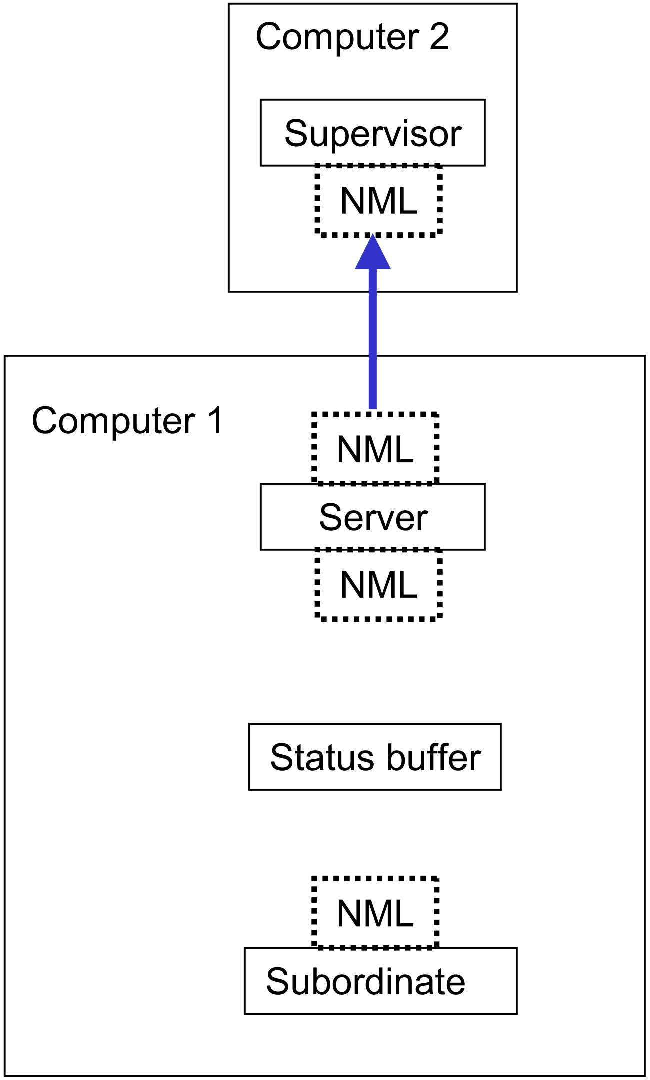
Remote Read Operation, Polling

Immediately afterward, the read operation returns the message last sent from the server, which it has saved. Initially, this is the empty message.



Remote Read Operation, Polling

Meanwhile, the server reads the contents of the status buffer, locally.

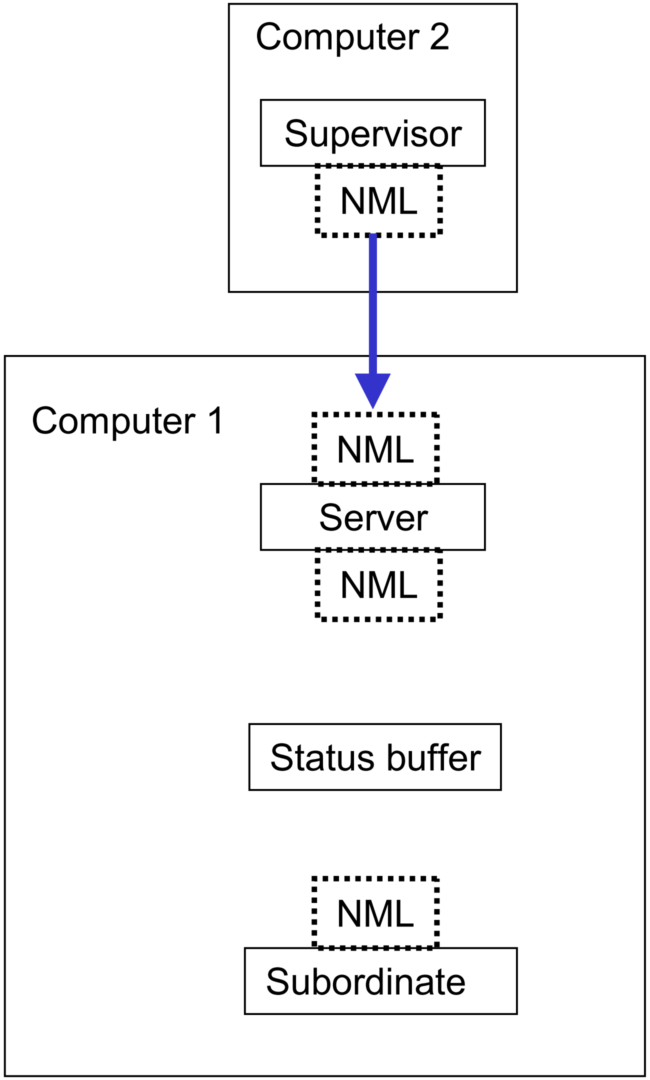


Remote Read Operation, Polling

Finally, the server sends the contents of the status buffer across the network to the supervisor. The NML code in the supervisor saves this to be returned immediately to the supervisor upon its next read.

The polling effect reduces the time spent in the read operation, at the expense of increased latency of data (roughly doubled).

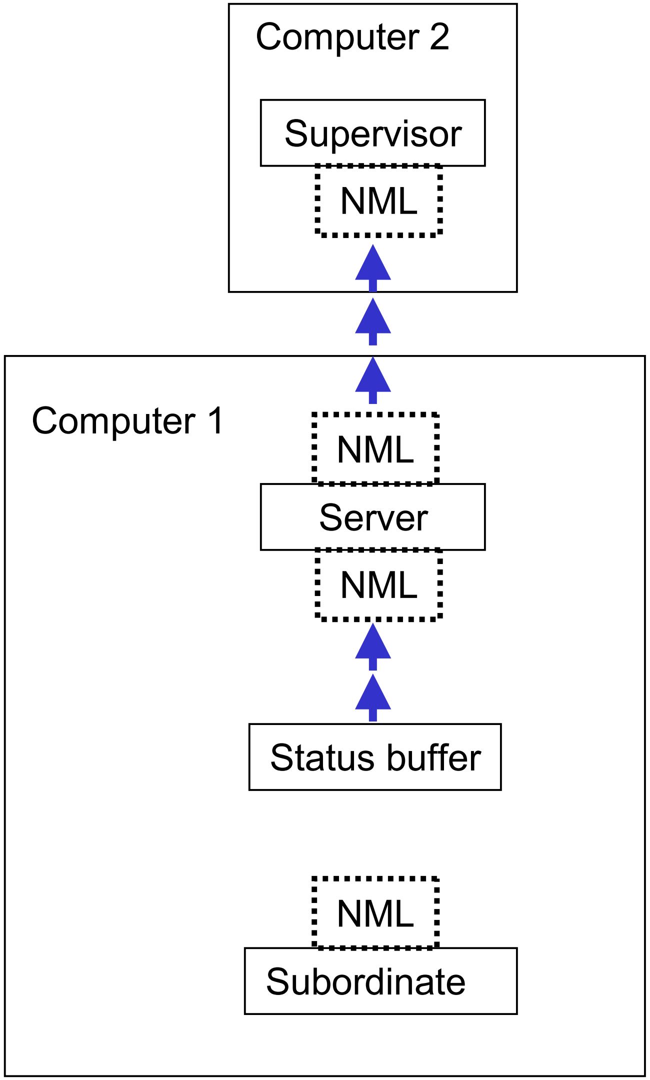
Processes should read regularly, since the returned data is the result of previous read requests.



Remote Read Operation, Subscribing

The remote reader (supervisor) sends a request to establish a subscription for the target buffer (status buffer) to the server, at a specified time interval.

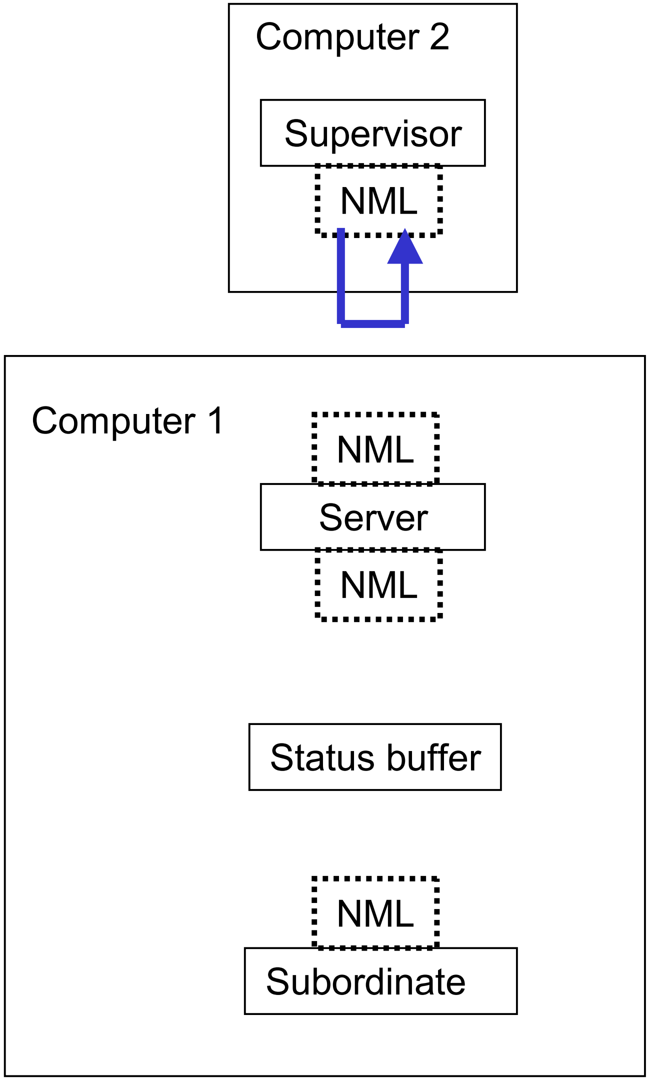
This is done automatically when the supervisor connects to the buffer, by the code in the NML constructor.



Remote Read Operation, Subscribing

The server reads the status buffer locally, at the specified time interval, and sends the buffer contents to the supervisor across the network.

The NML code in the supervisor saves the most recent message from the server.



Remote Read Operation, Subscribing

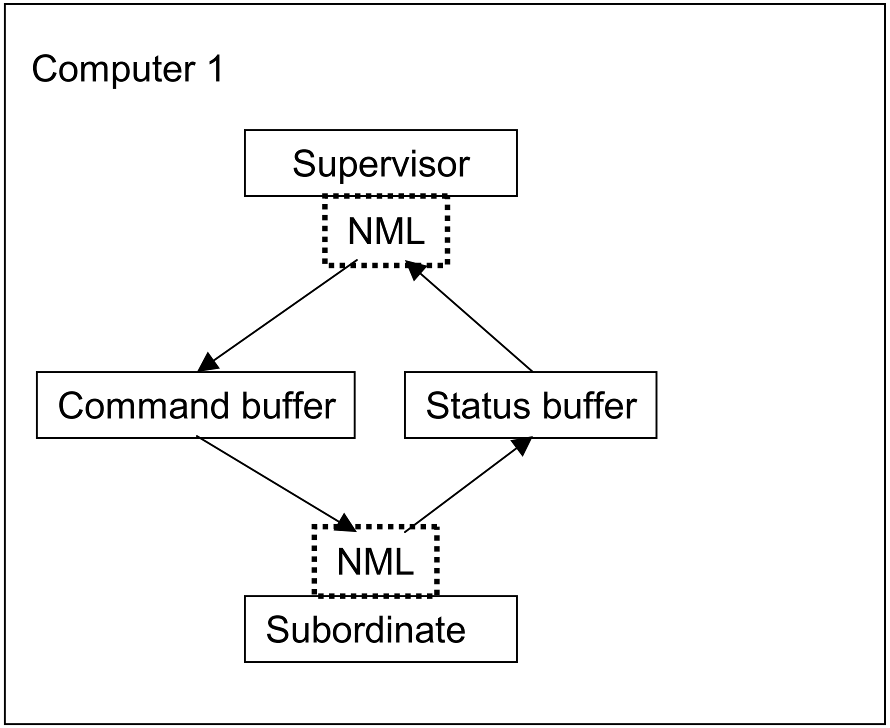
Read operations by the supervisor return the most recent saved value from the subscription.

The effect is to reduce the time spent in the read operation, since no network request is made.

Network traffic is halved, for the same reason.

The remote process should read at least as often as the subscribed data arrives, to prevent any network data queue from overflowing.

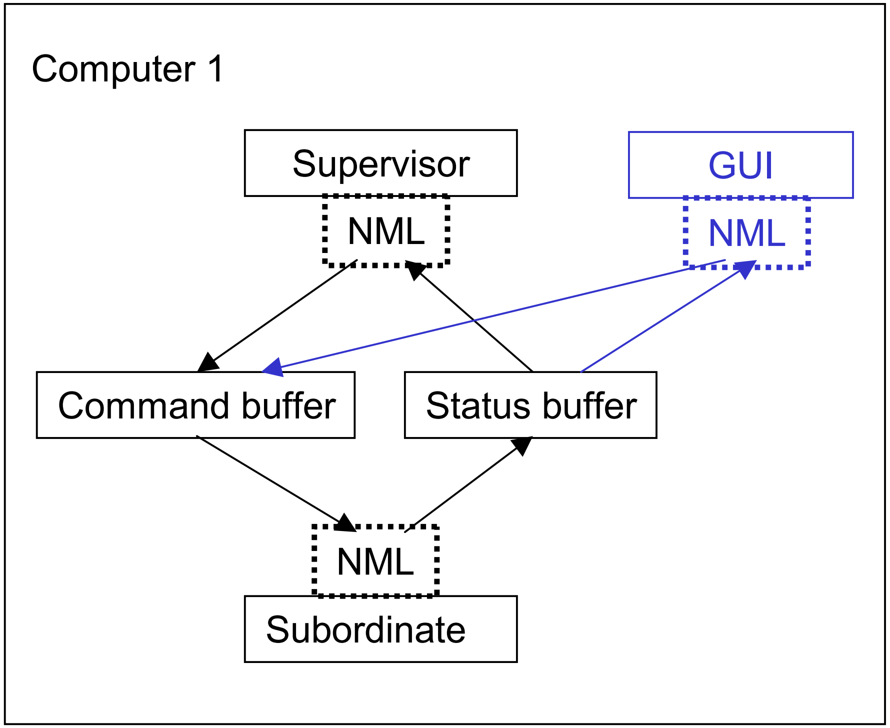
Example Configuration: Single Reader/Writer with Local Access



There are two NML buffers, each with a single reader and a single write. The supervisor writes the command buffer, and the subordinate reads it. The supervisor reads the status buffer, and the subordinate writes it. NML ensures mutual exclusion for data consistency.

```
# single.nml
# NML file for single reader, single writer, local access

# Buffers

# Name    Type  Host  Size Neut? (old) Buffer# MP Key
B command SHMEM comp1 256  0     0     1       2  1001
B status  SHMEM comp1 1024 0     0     2       2  1002

# Processes

# Name  Buffer   Type   Host  Ops Server? Timeout Master? Cnum

P subr  command  LOCAL  comp1 R   0       1.0     1       0
P subr  status   LOCAL  comp1 W   0       1.0     1       0

P supv  command  LOCAL  comp1 W   0       1.0     0       1
P supv  status   LOCAL  comp1 R   0       1.0     0       1
```

Example Configuration: Multiple Readers/Writers with Local Access

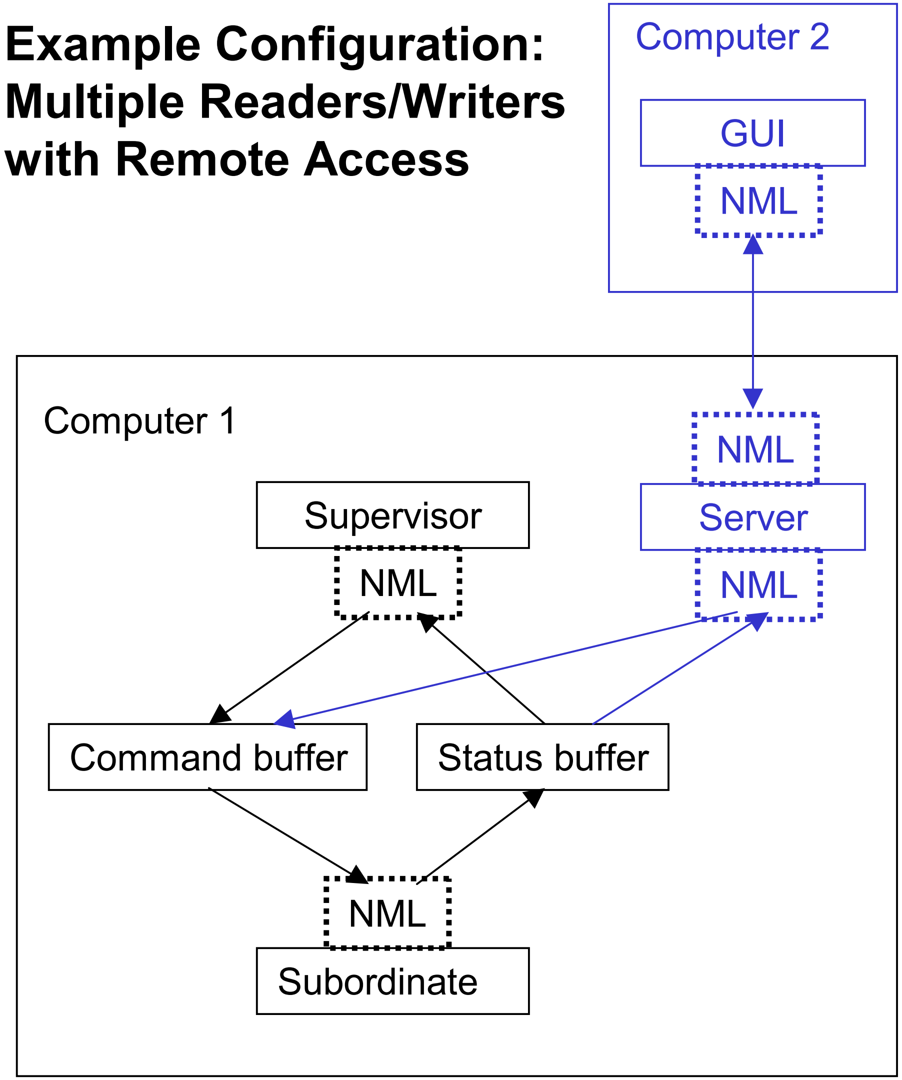


A graphic user interface (GUI) has been added. Now, the command buffer has two writers, and the status buffer has two readers. NML ensures mutual exclusion, as before.

Note that multiple writers need to coordinate their writing to avoid overwrites. Multiple readers of queued NML buffers need to coordinate their reading, to avoid stolen messages.

```
# multi.nml
# NML file for multiple readers-writers, local access

# Buffers

# Name    Type  Host  Size Neut? (old) Buffer# MP Key
B command SHMEM comp1 256  0     0     1       3  1001
B status  SHMEM comp1 1024 0     0     2       3  1002

# Processes

# Name  Buffer   Type   Host  Ops Server? Timeout Master? Cnum

P subr  command  LOCAL  comp1 R   0       1.0     1       0
P subr  status   LOCAL  comp1 W   0       1.0     1       0

P supv  command  LOCAL  comp1 W   0       1.0     0       1
P supv  status   LOCAL  comp1 R   0       1.0     0       1

P gui   command  LOCAL  comp1 W   0       1.0     0       2
P gui   status   LOCAL  comp1 R   0       1.0     0       2
```

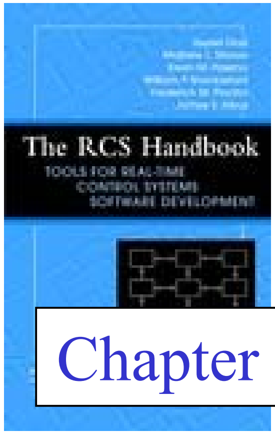
Example Configuration: Multiple Readers/Writers with Remote Access



GUI has moved to Computer 2. System now requires an NML server on Computer 1, which mediates read and write requests from the GUI. Communication between the GUI and the server is accomplished using TCP/IP sockets. The server accesses the target command and status buffers locally, as the GUI did before.

```
# remote.nml
# NML file for multiple readers-writers, remote access

# Buffers

# Name    Type  Host  Size Neut? (old) Buffer# MP Key
B command SHMEM comp1 256  0     0     1       4  1001 TCP=5001
B status  SHMEM comp1 1024 0     0     2       4  1002 TCP=5001

# Processes

# Name  Buffer   Type   Host  Ops Server? Timeout Master? Cnum

P subr  command  LOCAL  comp1 R   0       1.0     1       0
P subr  status   LOCAL  comp1 W   0       1.0     1       0

P supv  command  LOCAL  comp1 W   0       1.0     0       1
P supv  status   LOCAL  comp1 R   0       1.0     0       1

P gui   command  REMOTE comp2 W   0       1.0     0       2
P gui   status   REMOTE comp2 R   0       1.0     0       2

P serv  command  LOCAL  comp1 W   1       1.0     0       3
P serv  status   LOCAL  comp1 R   1       1.0     0       3
```

RCS Derived Messages



Chapter 6

- NMLmsg is the base message class
- rcs.hh defines two derived message classes
  - RCS_CMD_MSG, includes
    - serial_number
  - RCS_STAT_MSG, includes
    - echo_serial_number
    - status (RCS_DONE, RCS_EXEC, RCS_ERROR)

RCS Command Example

```
#include "rcs.hh" // declarations for RCS_CMD_MSG, etc.

#define MY_MOVE_TYPE 201

class MY_MOVE: public RCS_CMD_MSG
{
public:

  // constructor calls RCS_CMD_MSG base class with type and size
  MY_MOVE() : RCS_CMD_MSG(MY_MOVE_TYPE, sizeof(MY_MOVE)) {};

  // update function, which calls CMS's builtins
  void update(CMS *);

  // user-defined data
  // serial_number inherited from RCS_CMS_MSG
  float x, y, z; // commanded position
  float r, p, w; // command orientation
  float v; // commanded speed
};
```

RCS Status Example

```
#include "rcs.hh" // declarations for RCS_STAT_MSG, etc.

#define MY_STAT_TYPE 301

class MY_STAT: public RCS_STAT_MSG
{
public:

  // constructor calls RCS_STAT_MSG base class with type and size
  MY_STAT() : RCS_STAT_MSG(MY_STAT_TYPE, sizeof(MY_STAT)) {};

  // update function, which calls CMS's builtins
  void update(CMS *);

  // user-defined data
  // status, echo_serial_number inherited from RCS_STAT_MSG
  float x, y, z; // actual position
  float r, p, w; // actual orientation
  float v; // actual speed
};
```

RCS Derived Buffers

- NML is the base buffer class
- rcs.hh defines two derived buffer classes
  - RCS_CMD_CHANNEL, for RCS_CMD_MSG types
  - RCS_STAT_CHANNEL, for RCS_STAT_MSG types
- C++ example:

```
RCS_CMD_CHANNEL * buffer = new
   RCS_CMD_CHANNEL(formatFunction,
   "command", "supv", "config.nml");
```

Cloning Code

- Goal: replicating a complete NML application many times
- Example:
  - two vehicles, each with same NML-based control software
  - remote operator station
- One solution:
  - each vehicle has identical code
    - same executables
    - same NML configuration files
  - remote operator station has additional NML file
    - lists all buffers for all vehicles
    - buffer names modified to reflect different vehicle names

```
# vehicle.nml
# NML file for replicated vehicle controller, same for all

# Buffers

# Name    Type  Host  Size Neut? (old) Buffer# MP Key
B command SHMEM dummy 256  0     0     1       4  1001 TCP=5001
B status  SHMEM dummy 1024 0     0     2       4  1002 TCP=5001

# Processes

# Name  Buffer   Type   Host  Ops Server? Timeout Master? Cnum

P supv  command  LOCAL  dummy W   0       1.0     1       0
P supv  status   LOCAL  dummy R   0       1.0     1       0

P serv  command  LOCAL  dummy W   1       1.0     0       1
P serv  status   LOCAL  dummy R   1       1.0     0       1

# other local processes (subordinates) follow similarly
# remote.nml
# NML file for remote operator station for many vehicles

# Buffers-- one pair for each vehicle

# Name    Type  Host  Size Neut? (old) Buffer# MP Key
B cmd1    SHMEM real1 256  0     0     1       4  1001 TCP=5001
B stat1   SHMEM real1 1024 0     0     2       4  1002 TCP=5001
B cmd2    SHMEM real2 256  0     0     1       4  1001 TCP=5001
B stat2   SHMEM real2 1024 0     0     2       4  1002 TCP=5001

# Processes-- one process, many buffers

# Name  Buffer   Type   Host  Ops Server? Timeout Master? Cnum

P oper  cmd1     REMOTE dummy W   0       1.0     0       3
P oper  stat1    REMOTE dummy R   0       1.0     0       3

P oper  cmd2     REMOTE dummy W   0       1.0     0       3
P oper  stat2    REMOTE dummy R   0       1.0     0       3
```

Cloning Code (cont)

- Example:
  - two payload arms, each with same NML-based control software, on same platform
  - single task coordination controller on same platform
- Problem:
  - payload controllers need different configuration file entries to resolve resource conflicts, e.g., SHMEM keys
- One solution:
  - single configuration file, with multiple buffer names, process names
  - each controller is written to accept buffer names and process name as run-time arguments

```
# twoarm.nml
# NML file for dual payload arms and local task controller

# Buffers-- one pair for each arm

# Name    Type  Host  Size Neut? (old) Buffer# MP Key
B cmd1    SHMEM comp1 256  0     0     1       2  1001
B stat1   SHMEM comp1 1024 0     0     2       2  1002
B cmd2    SHMEM comp1 256  0     0     1       2  1003
B stat2   SHMEM comp1 1024 0     0     2       2  1004

# Processes-- arm (two styles), and task controller

# Name  Buffer   Type   Host  Ops Server? Timeout Master? Cnum

P arm   cmd1     LOCAL  comp1 R   0       1.0     1       0
P arm   stat1    LOCAL  comp1 W   0       1.0     1       0
P arm   cmd2     LOCAL  comp1 R   0       1.0     1       0
P arm   stat2    LOCAL  comp1 W   0       1.0     1       0

P task  cmd1     LOCAL  comp1 W   0       1.0     0       1
P task  stat1    LOCAL  comp1 R   0       1.0     0       1
P task  cmd2     LOCAL  comp1 W   0       1.0     0       1
P task  stat2    LOCAL  comp1 R   0       1.0     0       1
```

Payload NML Initialization

```
#include "nml.hh"
#include "myvocab.hh" // user NML vocabulary

// provide names of cmd, stat buffer as args
int main(int argc, char *argv[])
{
  // connect to NML buffers
  NML * command_buffer = new NML(my_format, argv[1],
       "arm", "myapp.nml");
  NML * status_buffer = new NML(my_format, argv[2],
       "arm", "myapp.nml");

  // rest of control
}
```

Alternatives to NML Initialization

- Other NML constructor parameters can be passed as arguments
  - process name
    - usually done in conjunction with variable buffer name
    - often done to reflect changing between local and remote execution
  - configuration file
    - often done to reflect application moving between demonstration platforms, with different host names, network ports
- Scripts can vary these arguments, resulting in flexible configuration without recoding or recompiling
- Applications can also be run in different directories, each with configuration file of same name but different contents

C Access to NML

- Currently no C interface to NML exists
  - inheritance was important to original design
    - e.g., class NML is parent of RCS_CMD_MSG, RCS_STAT_MSG
    - C interface needs to explicitly include base class members
  - function overloading was also important
    - e.g., CMS update functions update(float f), update(int i)
    - C interface needs one function per data type, e.g., update_float(float f), update_int(int i)
- C interface is more cumbersome, but can be added
  - code generation tool will help

C Linkage

- C applications can use NML through C linkage
  - NML portion of application is written in C++
  - link points are given C linkage
  - remaining application calls NML through C link points
- Not a C interface to NML, but a C interface to a particular NML application
  - need C functions for creating, reading, writing, and removing NML buffers
  - technique also applicable to any language, e.g., FORTRAN, Pascal

```
// C++ link point code-- compile this with C++ compiler

#include "nml.hh"
#include "myvocab.hh" // user NML vocabulary

static NML * command_buffer;
static NML * status_buffer;
static MY_STAT_MSG * my_stat_msg;

// call this to initialize NML from C
extern "C" int init_app()
{
  // connect to NML buffers
  command_buffer = new NML(my_format, "command",
         "supv", "myapp.nml");
  status_buffer = new NML(my_format, "status",
         "supv", "myapp.nml");

  // set pointer to status data from read, for short
  my_stat_msg = (MY_STAT_MSG *) status_buffer->get_address();

  return (command_buffer->valid() ||
          status_buffer->valid()) ? -1 : 0;
}
```

```
// more C++ link point code

// call this to send a move command
extern "C" int send_move_cmd(float x, float y, float z)
{
  MY_MOVE move_cmd;

  move_cmd.x = x;
  move_cmd.y = y;
  move_cmd.z = z;

  return command_buffer->write(move_cmd);
}

// call this to read current position
extern "C" int get_position(float *x, float *y, float *z)
{
  if (MY_STAT_TYPE == status_buffer->read()) {
    *x = my_stat_ptr->x;
    *y = my_stat_ptr->y;
    *z = my_stat_ptr->z;

    return 0; // means new data
  }

  return -1; // means no new data, or error
}
```

```
/* C application code-- compile this with C compiler, link with C++
   library containing functions with extern "C" linkage */

/* declarations for C linkage to C++ functions; should put in header */
extern int init_app();
extern int send_move_cmd(float x, float y, float z);
extern int get_position(float *x, float *y, float *z);

int main()
{
  float x, y, z;

  /* connect to NML application */
  if (0 != init_app()) {
    return -1;
  }

  /* control loop here */
  for (;;) {
    /* read position from NML application */
    get_position(&x, &y, &z);

    /* do control */

    /* write command to NML application */
    send_move(x, y, z);
  }
}
```

Summary

- NML provides a uniform application programming interface to communication, with C++ and Java bindings. Features:
  - multiple reading, writing, queued or mailbox, network access
  - protocol independence, platform neutrality
  - source code portability
  - real-time performance
- Programmer’s jobs:
  - define NML vocabulary
  - run code generation tool
  - populate application code with NML functions (open, read, write)
  - partition application and write NML config file