C++ Cross-Process Shared Memory Tools

The C++ Cross-Process Shared Memory Tools is new In PicklingTools 1.4.0. There have been some extensive updates as of PicklingTools 1.6.2.

Although the PicklingTools library has had tools to handle cross process shared memory for sometime (since 1.0.0), this release introduces some simple abstractions to help make using shared memory a little bit easier.

Reminder

First, a quick reminder: C++ Vals *can* be used with shared memory: this is the reason the `Allocator` became an inherent part of the PicklingTools back in 1.0.0. For example, to create a Tab in some shared memory region:

```
char* mem = ... create/attach shared memory across processes ...
StreamingPool *shm=StreamingPool::CreateStreamingPool(mem, bytes, 8);
Val v = Shared(shm, Tab());
v["a"] = "hello";   // Table and keys and values in shared memory
```

The shared memory can be created with `SHMCreate` or connected to with already created memory with `SHMAttach`, but it's tricky to get this right with the basic tools provided by the simple abstractions in "sharedmemory.h":

```
size_t bytes = 1024*1024;
char* mem = SHMCreate("shm_region", bytes);
// Created, but is it available yet?  Do you have to check
// if the entire shared memory has been mapped into the process
// with SHMInitialized, etc.
```

The process for using shared memory is a little clumsy: Who creates and who attaches? The creator is responsible for calling `SHMInitialize` and the user is responsible for called `SHMInitialized` to see if the region is ready, even it already mapped it. But these have to done in the right order, and it's not well documented.

To address these concerns, there are three new abstractions that handle Vals in shared memory a little better.

SHMMain, ServerSide and ClientSide

There are three new classes that make using shared memory a lot easier: SHMMain, ServerSide and ClientSide. These all come from:

```
#include "shmboot.h"   // Gets defn of SHMMain, ClientSide and ServerSide
```

SHMMain

SHMMain is responsible for creating the shared memory region of the proper size that all ServerSide and ClientSides can use. And that's it. It needs to be called exactly once.

Ideally, the SHMMain gets called exactly once in a startup process that gets called before everyone else: frequently at the start of some main process that has to be started before anything else in the app:

```
#include "shmboot.h"

```

```
int main (int argc, char**argv)
{
   int bytes = atoi(argv[1]);

   // Initial set-up code
   bool debug = true;
   SHMMain mem("shm_region", bytes, debug);
   mem.start();  // Actually calls and creates region

   ... start rest of application, fork processes, etc. ...

}
```

Sometimes, if an application is built piecewise, the model described above won't quite work: imagine something similar to a UNIX pipeline where each process may be communicating piecewise to server and a client. In a case like that, the "head" of the pipeline would be where the SHMMain should have to be created.

Once the SHMMain is created (and started), this establishes a shared memory zone or "pool" where queues and Vals can be used. Note that the SHMMain can be created from any process, even if there are no servers or clients in it: its sole purpose is to create and establish the shared memory region that later ServerSides and ClientSides use.

Expert Notes: If you are creating a system where the clients and servers all are all created from one main process, then that makes things easier: you don't *have* to specify a region, as all clients/servers that inherit from the main process can use the region already mapped in the main process: assuming all clients/servers `fork` from the main above and don't `exec`, the shared memory set-up from SHMMain will be inherited (in a process sense, not in a OOP sense) and be in the same address space in all the clients. However, if (like all the examples in the baseline), the clients and servers are completely separate processes, they HAVE to be mapped to the same area: in this case, you want to specify where in memory to map:

```
// On a 32-bit i386 Linux machine, force into a pretty unused area
SHMMain mem("shm_region_32bit", bytes, debug, (void*)0xB0000000);

// On a 64-bit x86_64 Linux machine, force into pretty unused area
SHMMain mem("shm_region_64bit", bytes, debug, (void*)0x700000000000ULL);
```

Although the small standalone programs tend to work without forcing to use a particular region, larger "real" applications tend to need to be forced. Although the ServerSide and ClientSide *can* explicitly set the memory needed, usually you only want to do this in the SHMMain (which sets it up in the right region) and that forces all clients and servers to attach to the right region.

ServerSide

The ServerSide presents the abstraction of a server, although strictly speaking it can also be either side of a communication. The most important thing a ServerSide does is to create the pipe (in shared memory) which will be used for queuing and enqueueing.

To create a pipe, you have to give it a string name (in the example below, `pipename`) and start the ServerSide:

```
// Create a pipe in the shared memory region (string name used in SHMMain)
// The pipe has the given capacity (capacity is in packets).
bool debug = true;
int packet_capacity = 4;
```

```
ServerSide server("shm_region",
                  "pipename", packet_capacity. true);

server.start();   // pipe only created when started
CQ& pipe = server.pipe();
```

Once the pipe has been created, it can be enqueued or dequeued from: *Note that queueing and dequeueing is thread-safe*: I.e., you can have multiple processes operating on the queue simultaneously and its state is never inconsistent. Typically, the ServerSide enqueues and the ClientSide dequeues from the given CQ:

```
// ServerSide enqueues
for (int seq=0; ; seq++) {

   // Create the value to enqueue in shared memory
   Val data = Shared(server.pool(), Tab());  // Create a Tab in shared memory
   data["sequence_number"] = seq;

   // Now enqueue
   bool enqueued = false;
   real_8 timeout_in_seconds = 3.2;
   while (!enqueued) {
       enqueued = pipe.enqueue(data, timeout_in_seconds);
       if (!enqueued) {
           cerr << "Failed to enqueue after " << timeout_in_seconds
                << " in seconds ... trying again ..." << endl;
       } else {
           cout << "Enqueued! Going to next packet" << endl;
           break;
       }
   }

}
```

Note that the enqueue has a timeout: in case the queue is full and the Val can't be placed in the queue, it will wait up to `timeout_in_seconds` for some space to open up. If the space opens up in the given time, the data is enqueued and enqueue returns true: the data has now been enqueued. If space doesn't open up in the given time, enqueue returns false. In the example code above, the sender just keeps retrying, but gives a warning every 3.2 seconds warning indicating the queue is full.

Although typically the ServerSide is used as the *send* side only, there's no reason the ServerSide can't dequeue as well (see below for examples of how to dequeue). This ability might be useul if that queue needed to be cleaned as part of a shutdown or restart.

There is another call to enqueue data `pipe.enq(data)` which will block forever; this is simpler and will work but makes it harder to detect error conditions and/or gracefully exit.

ClientSide

The ClientSide is the other end of the pipe: if the ServerSide creates the pipe and writes to it, then the ClientSide waits for the pipe to be created so it can attach and read from the pipe:

```
 ClientSide client("shm_region", "pipename", true);
 client.start(); // blocks waiting for pipe to be created

 CQ& pipe = client.pipe();
```

```
// Once pipe is available, we can read from it
real_8 timeout_in_seconds = 5.5;
while (1) {

  // Try to dequeue a single packet
  Val packet;
  bool valid=false;
  while (!valid) {

      valid = pipe.dequeue(timeout_in_seconds, packet);
      if (!valid) {
            cerr << "Couldn't dequeue after " << timeout_in_seconds
                 << " ... trying to dequeue again ... " << endl;
            continue;
      } else {
            cout << "Got packet!" << endl;
            break;
      }

  }
}
```

Note that the client (in order to read from the proper pipe) has to match *both* the name of the shared memory region *and* the pipename the server writes to.

In the above example, if the client can't get something from the pipe immediately, thee queue blocks for up to a few seconds. If something appears on the pipe before the timeout, then that value is dequeued and taken off the pipe and valid becomes true. If after those few seconds the pipe is still empty, valid becomes false and the dequeue fails.

This is basic paradigm: Use SHMMain to create the shared memory region before anything else starts, start and create a sender with a ServerSide and client with ClientSide.

Note that the ClientSide and ServerSide will block until SHMMain is called and creates the memory pool. The ClientSide will block and wait until the the ServerSide creates the pipe. The only way this can become problematic if the shared memory pool has a "unclean" shutdown: if you can't guarantee SHMMain is called before all the clients and servers are created, it's important to make sure the shared memory region has been destroyed, otherwise the client and server may pick up the shared pool from the LAST invocation. To make sure the shared memory is clean, make sure /proc/shm does has been cleaned up and does NOT contain the memory pools; this is discussed a bit more in the *Five Major Headaches* section below later.

When the Val is dequeued from the pipe, it is copied out using the copy constructor of the Val. So, Vals with Proxys simply increase/ decrease reference counts. Prior to PicklingTools 1.6.0, the value was left in the pipe until it was overwritten by later puts to the pipe. As of PicklingTools 1.6.0, when a value is dequeued, the value is copied out and then the Val in the pipe is immediately destructed. Why is this important? If your Val uses a lot of shared memory, letting a copy live in the pipe after a dequeue wastes resources. By immediately destructing after the dequeue copies it out, its resources are immediately released.

Middleside

Many times a server is also a client: it reads from one queue and posts to another queue (like a UNIX pipeline): this role is frequently called a *transformer* in X-Midas or M2k speak. The example below show how to *both* read from a client and post to a server in the same process:

```
ClientSide client("shm_region", "pipe_0", true);
client.start(); // read from this pipe
CQ& input = client.pipe();

ServerSide server("shm_region", "pipe_1", 4, true);
server.start(); // write to this pipe
CQ& output = server.pipe();

while (1) {

    // Read from input pipe
    Val in_packet;
    if (!input.dequeue(.1, in_packet))
        continue; // retry to get input

    // Write to output pipe
    bool enqueued = false;
    while (!enqueued) {
      enqueued = out.enqueue(in_packet, .1);
    }
}
```

In this way, a client and server can be used together. In fact, there is no apriori limit on the number of clients and servers that can be in a single process (limited only by the amount of memory). Multiple clients and/or servers allows programming any type of semantics you want: see the upcoming section.

Timeouts and External Shutdown Conditions

The SHMMain (and ServerSide and ClientSide) all take the same last two (normally defaulted) arguments: an external break checker and microsecond sleep period:

```
SHMMain (const string& memory_pool_name, size_t bytes,
         bool debug=false, void* forced_addr=0,
         BreakChecker external_break=0, int_8 micro_sleep=int_8(1e5))
```

Some of the internal routines to the SHMMain (and ServerSide and ClientSide) have loops where they are waiting for things to happen (shared memory to come up, first time it's zero filled, wait for that, etc). In case there are problems, there is a systematic way to monitor a global shutdown conition and shutdown an application cleanly. The *external_break* is simply a typedef for a pointer to a function:

```
typedef bool (*BreakChecker)();
```

If the user has somehow set an external condition that signifies "time to shutdown", the user can wrap that check in a routine and pass it to the SHMMain. How often that external condition is checked is the next arguments *micro_sleep*: How many microseconds we sleep before we check (a) coming up conditions and (b) external break. You have to be a little careful with this: if you set it too big, the SHMMain will come up quickly, but there will be a lot of polling (potentially taking too much CPU). If you set it too small, it will take longer to come up and you will might miss external shutdown conditions. The default, 1e5 in microseconds (thus, .1 seconds) is a fairly reasonable compromise. By default, there is no break checker: it's a hook the user can fill in.

Example: In X-Midas, there is a global flag called *Mc->break*. When an application is coming down (because contrl-C was hit, or an error happened), the *Mc->break* is set for about 5 seconds to tell all primitives to come down. The external break checker for X-Midas would look something like:

```
// Wrap the break checker into a pointer to function we can call
inline bool Mc_break_checker ()
{
  // memory barrier before fetch: may not need ... may be able to comment
  // out if we don't have gcc atomics ...
  __sync_synchronize();

  volatile bool_4* volatile mcb = &Mc->break_;
  volatile bool_4 br = *mcb;
  if (br) {
    return true;
  } else {
    return false;
  }
  // return Mc->break_; // this is volatile, but only this routine needs that!
}
```

It's a simply hook that allows an application to get out if there are problems; without this check, it's possible for an app to "hang" waiting for something to happen. Another possible break checker would be: if a total of n seconds have elapsed and the app still hasn't come up, kill the app.

Complex Interactions

By default, the CQ class (so-called because it is a Queue that preserves thread-safety using a Condition variable: CQ) has a simple interface: Vals (which exist in shared memory, this is critical!) can either be enqueued or dequeued. Period. There can be any number of processes enqueueing or dequeueing simultaneously, but (this is important) *there is no notion of multicast*: once a Val is dequeued by any reader, it's gone, even if there are twelve readers. If we wanted to support multicast, or any complicated semantics for multiple readers, we have to enforce those ourselves.

For example, if we wanted to have the multicast semantics to only dequeue when all the readers have read the packet (or anything more complicated), we could put all the logic in a simple component with one input and multiple outputs:

```
// Implement a simple multicast with n clients where
// all clients see all data that is dequeued.

// Start up the single input
ClientSide single_input("shm_region", "pipe_input", true);
single_input.start();
CQ& input = single_input.pipe();

// Array of outputs: each output queue will see the input exactly once
Array<ServerSide*> outputs;
for (int ii=0; ii<number_of_outputs; ii++) {
   ServerSide* ssp=new ServerSide("shm_region", "out"+Stringize(ii), 4, true);
   ssp->start();
   outputs.append(ssp);
}

while (1) {

   // Pull input off the input queue
   Val in_packet;
   if (!input.dequeue(.1, in_packet)) {
      continue;  // nothing available, try again
```

```
   }

   // Got input: implement multicast semantics so
   // all outputs see the same input
   for (int ii=0; ii<outputs.length(); ii++) {
      ServerSide* ssp = outputs[ii];
      CQ& out = ssp->pipe();

      // Try to enqueue; blocks until delivered
      out.enq(in_packet);
   }
}
```

With the example above, all outputs see a copy of the input. Since *hopefully* the input packet is just a proxy (where the underlying Tab is shared), this should be a quick and easy dispersal.

Although this abstraction makes it a little harder to implement multiple readers and writers, it gives you full mechanism to implement any policy for multiple readers. For example, in the example above, the multicast can get stuck if one the readers never reads its pipe: the enqueue blocks forever. What if we wanted data to drop if the reader hadn't read it after 5 seconds?:

```
bool enqueued = out.enqueue(in_packet, 5.0);
if (!enqueued) {
   cerr << "Reader " << ii " << is too slow, dropping packet" << endl;
   continue;
}
```

Thus with a simple change, we have implemented a custom multi-cast semantics. For systems where dropped data is okay, we can implement whatever semantics we need with whatever time constraints are relevant. If dropped data is not okay, we can keep retrying to send data, with some messages. We could also implement a nice GUI to show the status of a pipe. Whatever is needed can be built on ServerSide, ClientSide and timeouts.

Address Randomization

Many Linuxes today implement the Address Randomization "feature" to stop hackers from exploiting address space regularity. In other words, the code and data part of your program can be randomly place "anywhere" in main memory. This can be problematic for systems where shared memory needs to be mapped in: What if regions conflict? What part of memory is used? To that end, you may have to force your processes to turn off this feature. This feature can be turned off on a per-process basis easily:

```
% setarch i386 -L -R serverside_ex ...    # 32-bit machine
       or
% setarch x86_64 -L -R serverside_ex ...  # 64-bit machine
```

Frequently, the simple examples work without the above, but the more complex programs may need to do the above. Note that the examples remind you to use setarch when you run them.

If you forget to turn off the feature, then each of the SHMMain, ClientSide and ServerSide will *warn* you with a large message to standard error:

```
% serverside_ex ...   # forgot to run with setarch!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! WARNING !!!!!!!!!!!!!!!!!!!!!!!!
! It appears that the address randomization feature is still on.
```

```
! Your SHMMain/ServerSide/ClientSide is unlikely to work correctly.
! Program will continue running ... but may not run correctly ...
!
! Make sure the process that's gets started up has this feature
! turned off using setarch.  For example:
!  % setarch i386 -L -R startup_program     # 32-bit machine
!           or
!  % setarch x86_64 -L -R startup_program   # 64-bit machine
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Detecting the absence of the Address Randomization feature is a bit tricky, so you may be using setarch correctly but it still outputs the warning. Lots of false positive warning messages are clumsy and messy, so there is a mechanism to turn off the potential warning. For example, to turn off this message for SHMMain:

```
SHMMain mem(...);
mem.warning(false);  // Turn off warning message above
mem.start();         // Warning message above suppressed
```

Both ClientSide and ServerSide have the same method. By default, the warning is turned on: the idea is that it is better to get a warning message when you are first starting so you can figure out how things work; Once you are comfortable and always turning on the setarch feature, you don't need the warning anymore and can turn it off if needed.

Examples

There are three standalone examples and three X-Midas examples in the PicklingTools baseline demonstrating the shared mem client/server tools.

In the C++ area are serverside_ex.cc, middleside_ex.cc and clientside_ex.cc examples. These three examples should be run together *on the same machine*:

```
# In one shell prompt
% setarch i386 -L -R serverside_ex mempool 1000000 pipe1 4
```

The `serverside_ex` creates the shared memory region (called `mempool`) of one million bytes. The pipe the server will write to is `pipe1` and it will have a capacity of four packets:

```
# In another shell prompt
% setarch i386 -L -R middleside_ex mempool pipe1 pipe2 4
```

The `middleside_ex` waits for the shared memory region (called `mempool`) to be available. Once it is, the middleside reads from `pipe1` and writes to `pipe2`; `pipe2` also has a capacity of four packets:

```
# In yet another shell prompt
% setarch i386 -L -R clientside_ex mempool pipe2
```

The `clientside_ex` waits for the memory pool, waits from `pipe2` to be created, then reads from `pipe2`.

This will cause the server to talk the middle and the middle will talk to the final client. Note that SHMMain is created *exactly once* by serverside_ex.

Also note that the X-Midas primitives can talk to the standalone executables.

Five Biggest Headaches

The five biggest problems getting shared memory across platforms to work are:

1. Memory size: Most Linuxes are constrained by how much shared memory they can allocate. If the allocation is too big, then Linux will simply fail. Carefully try larger and larger sizes of shared memory from SHMMain to make sure that your box can legally create and use that much shared memory.
2. Using Vals in shared memory. Once a Val is created in shared memory, most updates on the table will cause the new keys/values to be created in the same shared memory. If you wish to enqueue a table or Array, make *sure* the entire table is in shared memory:

   ```
   // Okay
   Val data = Shared(shm, Tab());  // created in shared memory
   pipe.enqueue(data);             // Okay, because data in shared memory

   // !!!! NOT OKAY!!!
   Val data2 = Tab();  // data2 NOT in shared memory!!
   pipe.enqueue(data2);  // Will seg fault
   ```

   There is a routine called `IsSHM` from #include "checkshm.h" which allows the user to check and see if a Val is completely contained in shared memory. In debug mode, this is a very useful tool; before data is enqueued on a shared memory pipe, the table can be checked to make sure all of its parts are in shared memory.
3. Left-Over files. By default, the shared memory regions have an file in the /dev/shm area. There is good news and bad news about this. If you do not destruct SHMMain (in the C++ sense), then the region persists. This can be good because your queues can persist across time: assuming a client connect and disconnects frequently, this can just work as the new connection will simply pick up where the last one left off.

   This can be horrible if the queues get into an inconsistent state: then every client will simply break.

   It can be useful to completely clean /dev/shm of your shared memory. By default, when you create shared memory region like the "shm_region" in all the examples above, two files are created:

   ```
   /dev/shm/shm_region_boot
   /dev/shm/shm_region
   ```

   The boot region is very small and used only to pass global information to each client (where memory should be mapped, the size of the memory, where to find the pipes, etc). The boot is first mapped in anywhere in memory. The data in the boot informs where to map the main section: the main section contains the giant memory pool.

   To make sure your application starts in a fresh start, it's probably worth removing `/dev/shm/shm_region_boot` and `/dev/shm/shm_region` before starting, or restarting your application. Note that *by default*, SHMMain will completely clean-up for you when you create and start the SHMMain component.
4. Forgetting About Address Randomization

   Frustratingly, sometimes things will work with the address randomization on, then one small change will cause everything to stop working. It's best put this as part of a start-up script where all processes will "inherit" this attribute:

   ```
   % setarch i386 -L -R startup_process
   ```

   See the *Address Randomization* section above.

5. Holding On Too Long or Letting Go Too Soon

   What does `Shared(client.pool(), Tab())` return? A proxy to a Tab in shared memory protected by a process-safe reference count. Remember, that every copy of the proxy increments the reference count:

```
// Serverside
{
  Val v = Shared(shm, Tab());  // ref count at 1
  pipe.enq(v);                 // ref count at 2
} // Val destructed, ref count at 1


// Clientside
{
  Val off = pipe.deq();  // Ref count at 1
}  // Val destructed, ref count at 0, memory reclaimed
```

   Thus, once you have enqueued your shared Tab, you can let go of the packet and let the client dequeue it. Once the client dequeues it and is done, it will be reclaimed by the pool.

   Be careful not to keep your Tab alive after you have enqueued it, or that could become a memory growth.

Tips and Tricks

Here's a collection of tricks and tips that can make your life easier.

1. You can always ask any of the SHMMain, ClientSide or ServerSide for it's allocator:

```
ServerSide server_side(...);
Allocator *a = server_side->pool(); // Get the allocator
```

   That way you can be assured to get the right allocator.

2. You can ask a Val for it's allocator:

```
Val v = ...;
Allocator *a = v.allocator();
```

   If the value was from a SharedPool, it will give you access to that. If this Val was using the "standard" heap, a will be NULL.

3. If you have an allocator, you can call allocate to get some memory from it, and deallocate to free it. If you are just using Vals/Tabs/etc. with it, you should be able to use the Shared or Protected stubs:

```
Val v = Shared(a, Tab());  // Give me a Tab from allocator a
```

Just a few tricks and tips. The SHMQ X-Midas option tree shows one way to use these routines to implement a shared memory system.

Conclusion

With three simple abstractions, tables across shared memory can be much easier to manipulate.

Known Bugs:

There are still some known bugs: we prefer to release early so as to get feedback, even if there are some known issues.

1. The `int_un` and `int_n` DO NOT work with shared memory. A future version will fix this. **FIXED: in PicklingTools 1.4.1**
2. Should a Val initialized from a Proxy copy the allocator? This makes the `IsSHM` check without an explicit work-around for Proxies.