MANTARO
PRODUCT DEVELOPMENT SERVICES

Tech Talk Presents

Efficiently Coding
Communications Protocols
in C++

About Tech Talk

Tech Talk is a series of white papers created by engineers at Mantaro to share our knowledge in different technical areas. With a diverse and progressive set of clients, Mantaro has a rich history of staying at the leading edge of the technology curve. To keep the entire engineering team updated on the latest tools, trends, and technology, Mantaro holds regular meetings where engineers share their findings. Tech Talk captures this wealth of knowledge in the form of white papers which are made available to the general pubic through our website. If you have any questions or comments, please email us at techtalk@mantaro.com.

About Mantaro

Mantaro provides a full range of product development services including analog, digital, and RF hardware design, FPGA design, embedded system development, and RFI/EMI design consulting. Our technical staff comprises of highly talented professional engineers with a history of successful product development and innovative design experience. We are an Altera Certified Design Center, Xilinx Alliance Program Member, Analog Devices DSP Third Party Developer, TI Low Power Wireless Third Party Developer, Cavium OCTEON Partner, WiPower Design Partner, Tilera Design Partner, Sparsense Design Partner, etc providing industry leading FPGA and hardware/software experience. Mantaro continues to grow by exceeding customer's expectations and acting as extensions of their engineering team.

Mantaro maintains its headquarters and lab facilities in our Germantown, Maryland location. Mantaro's lab environment fosters the continual professional development of our staff and allows us to leverage our collective development experience and established methodologies to reduce both the time and costs associated with delivering high quality products. Mantaro's processes are based on our commitment to excellence and grounded in maintaining a close partnership with our clients throughout all phases of a project.

Contact Information

Mantaro Product Development Services, Inc.

20410 Century Blvd, Suite 120

Germantown, MD 20874

301-528-2244

www.mantaro.com

techtalk@mantaro.com – comments about Tech Talk

sales@mantaro.com – information on Mantaro's services

info@mantaro.com   all other inquiries

Efficiently Coding Communications Protocols in C++

Harvey A. Sugar - **Hsugar@mantaro.com**

What is Efficient Coding?

Efficient coding could have two meanings. First efficient coding could refer to making the best use of a developer's time when coding and debugging and embedded application. Second efficient coding could refer to developing code that has high performance using the minimum of processor and memory resources when executing. This paper I will show how using C++ classes to implement a communications protocol can improve the developer's efficiency then I will discuss some techniques for improving the execution efficiency.

C++ and Development Efficiency

The C++ object model helps speed development in two ways. First through abstraction or information hiding, C++ allows us to divide and conquer complex applications like communications protocols. Information hiding is one of the most powerful tools we have for dealing with complexity. The C++ object model allows a developer to explicitly define what information is to be hidden through the use of private data. The C++ object model is far superior to C's module concept which is only enforced by programming conventions and not enforced by the language rules. C modules only work well if everyone follows the rules.
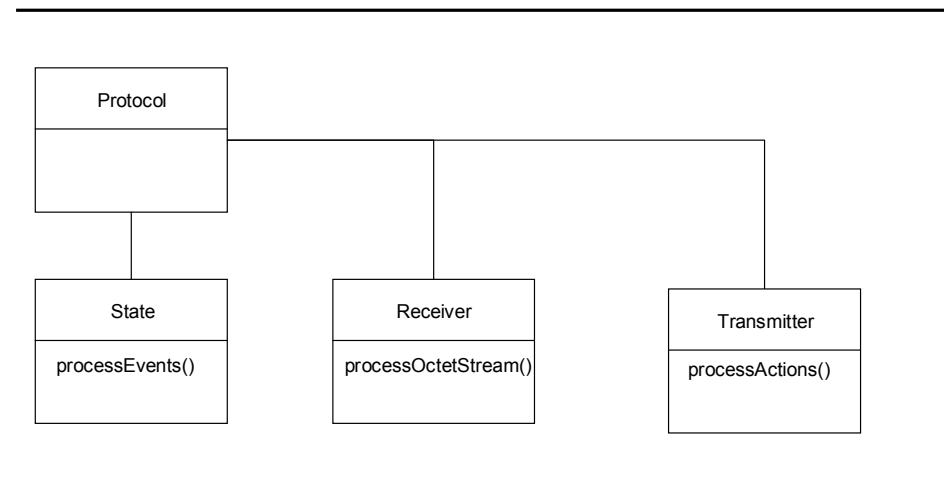
Second, we can use C++ classes to closely model the concepts defined in a communications protocol's specification. In the following sections we will see how a direct one-to-one mapping between the protocol specification and C++ classes and methods can be achieved. By using the same concepts and terminology in the software as is used in the protocol specification, the code becomes easier to implement, understand and maintain.

Protocol Layer Model

Communications protocol specifications usually contain three primary components: the message encapsulation or envelope, the message semantics and the protocol state. The figure below shows a class diagram for a protocol layer.

```
                    ┌─────────────────┐
                    │    Protocol     │
                    ├─────────────────┤
                    │                 │
                    │                 │
                    └────────┬────────┘
            ┌────────────────┼────────────────┐
  ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐
  │      State      │ │    Receiver     │ │   Transmitter   │
  ├─────────────────┤ ├─────────────────┤ ├─────────────────┤
  │ processEvents() │ │processOctetStream()│ │ processActions()│
  └─────────────────┘ └─────────────────┘ └─────────────────┘
```

The receiver is responsible for processing the incoming octet stream. It validates the message and removes the encapsulation. The receiver also acts as a de-multiplexer, passing messages to the upper layers or translating the messages into events that are processed by the state machine.

The state machine processes protocol events which may be specific messages from the lower layers or other types of events from the upper or lower protocol layers.

The transmitter implements actions requested by the state machine by translating them into protocol messages. The transmitter also encapsulates messages from the upper protocol layers and passes the encapsulated messages to the lower protocol layers.

From Models to Code – Implementing the Point-to-Point Protocol

Now we will look at a specific protocol, the Point-to-Point Protocol (PPP) as defined in RFC-1661. The Point-to-Point protocol is actually a family of protocols sharing the same encapsulation. A PPP implementation must include the Link Control Protocol (LCP) and one or more Network Control Protocols, one for each network protocol supported. The most common Network Control Protocol is the IP Control Protocol or IPCP. A PPP implementation may also include one or more authorization protocols. So the actual structure of the PPP layer is a bit more complex than the idealized structure shown above. The figure below shows the structure for the PPP protocol. It include of a single receiver and transmitter that implement the message encapsulation and message semantics. The LCP, each NCP and each authorization protocol include a context and their own state.
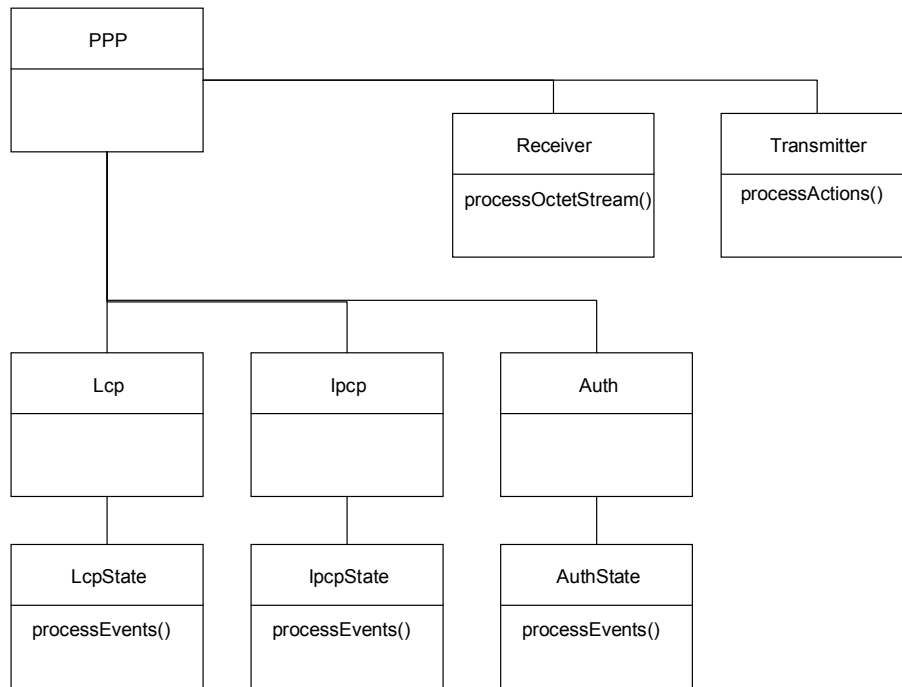
PPP LCP Details

Implementing the complete family of protocols that make up PPP is beyond the scope of this paper so we will concentrate on the LCP for our example. The following table shows a list of events and actions defined in RFC-1661:
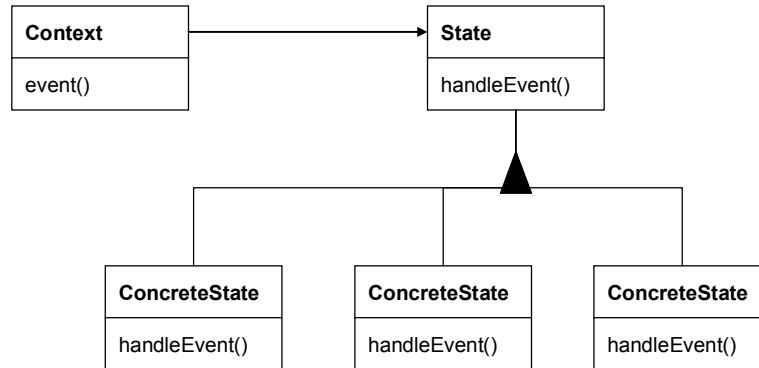
	Events			Actions
Up	Lower Layer is Up		Tlu	This layer up
Down	Lower Layer is Down		Tld	This Layer Down
Open	Administrative Open		Tls	This Layer Started
Close	Administrative Close		Tlf	This Layer Finished
TO+	Time Out with counter > 0		Irc	Init Retry Count
TO-	Time Out with counter = 0		Zrc	Zero Retry Count
RCR+	Recv Config Request (Good)		Scr	Send Config Request
RCR-	Recv Config Request (Bad)		Sca	Send Config Ack
RCA	Recv Config Ack		Scn	Send Config Nack
RCN	Revc Config Nack		Str	Send Terminate Request
RTR	Recv Terminate Request		Sta	Send Terminate Ack
RTA	Recv Terminate Ack		Scj	Send Code Reject
RUC	Recv Unknown Code		Ser	Send Echo Reply
RXJ+	Recv Code or Protocol Reject (Permitted)			
RXJ-	Recv Code or Protocol Reject (Catastrophic)			
RXR	Recv Echo Request or Reply or Recv Drop Request			

RFC-1661 also defines ten states for the PPP LCP; 0 - Initial, 1 - Starting, 2 - Closed, 3 - Stopped, 4 - Closing, 5 - Stopping, 6 - Request-Sent, 7 - Ack-Received, 8 - Ack-Sent, and 9 - Opened. With the events, actions and states defined, RFC-1661 goes on to define the protocol's behavior shown in the state table. The table's rows represent the events and the columns represent the states. The actions and next state are show for each event in each state. The empty cells are invalid events for that state.

Events	States									
	0	1	2	3	4	5	6	7	8	9
Up	2	Irc,src/6	-	-	-	-	-	-	-	-
Down	-	-	0	Tls/1	0	1	1	1	1	Tld/1
Open	Tls/1	1	Irc, scr/6	3	5	5	6	7	8	9
Close	0	Tlf/0	2	2	4	5	Irc,str/4	Irc,str/4	Irc,str/4	Tld,irc,str/4
TO+	-	-	-	-	Str/4	Str/5	Scr/6	Scr/6	Scr/6	-
TO-	-	-	-	-	Tlf/2	Tlf/3	Tlf/3	Tlf/3	Tlf/3	-
RCR+	-	-	Sta/2	Irc, scr,sca/8	4	5	Sca/8	Sca, tlu/9	Sca/8	Tld, scr,sca/8
RCR-	-	-	Sta/2	Irc, scr, scn/6	4	5	Scn/6	Scj/7	Scn/6	Tld, scr, scn/6
RCA	-	-	Sta/2	Sta/3	4	5	Irc/7	Scr/6	Irc, tlu/9	Tld, scr/6
RCN	-	-	Sta/2	Sta/3	4	5	Irc, scr/6	Scr/6	Irc, scr/8	Tld, scr/6
RTR	-	-	Sta/2	Sta/3	Sta/4	Sta/5	Sta/6	Sta/6	Sta/6	Tld, scr, sta/5
RTA	-	-	2	3	Tlf/2	Tlf/3	6	6	8	Tld, scr/6
RUC	-	-	Scj/2	Scj/3	Scj/4	Scj/5	Scj/6	Scj/7	Scj/8	Scj/9
RXJ+	-	-	2	1	4	5	6	6	8	9
RXJ-	-	-	Tlf/2	Tlf/3	Tlf/2	Tlf/3	Tlf/3	Tlf/3	Tlf/3	Tld, irc, str/5
RXR	-	-	2	3	4	5	6	7	8	Ser/9

The State Pattern

The table shows the complexity of the protocol's behavior and coding this state machine as C case statements or if-else statements is a daunting task. However, the protocol state is easily implemented as a number of simple C++ classes using the State Pattern described in *Design Patterns* by Gamma, et al. shown below.

Context	State
event()	handleEvent()

ConcreteState	ConcreteState	ConcreteState
handleEvent()	handleEvent()	handleEvent()

The Context object in the state pattern contains a pointer to a concrete State object. Events to be processed by the context are deferred to a handleEvent() method in the concrete State thus all event handling is determined by the context's current state.

The LcpContext and LcpState Base Class

The class declaration for the LcpContext is show below. Note that we have declared methods for every event and action defined by the protocol. The LcpState class is declared as a friend class so that it has access to private methods for executing actions and to change the LcpContext's state.

```cpp
class LcpContext
{
public:
    LcpContext(LcpState* initialState, PppTransmitter* transmitter);
    ~LcpContext();

    // Events
    void up();
    void down();
    void open();
    void close();
    void timeOut();
    void timeOutRetryExpired();
    void rcvConfigReqGood();
    void rcvConfigReqBad();
    void rcvConfigAck();
    void rcvConfigNakRej();
    void rcvTermReq();
    void rcvTermAck();
    void rcvUnknownCode();
    void rcvCodeProtRejPermitted();
    void rcvCodeProtRejCatastrophic();
    void rcvEchoReq();
    void rcvDiscardReq();

private:
    friend class LcpState;

    // Actions
    void thisLayerUp();
    void thisLayerDown();
    void thisLayerStarted();
    void thisLayerFinished();
    void initRetryCount();
    void zeroRetryCount();
    void sendConfigReq();
    void sendConfigAck();
    void sendConfigNakRej();
    void sendTermReq();
    void sendTermAck();
    void sendCodeRej();
    void sendEchoReply();

    void changeLcpState(LcpState* state);
    LcpState*        _state;
    LcpTimer*        _timer;
    PppTransmitter*  _transmitter;
};
```

The implementation of the event methods is simple. The event processing is simply deferred to the State object pointed to by _state.

```
void LcpContext::up()
{
    _state->up(this);
}
```

The action methods are also simple, deferring the processing to a transmitter or timer object. This decouples the State objects from the PppTransmitter, PppReceiver, and LcpTimer classes so that the state classes can be easily used for a number of PPP applications such as PPP over Ethernet (PPPOE) or Packet Over SONET (POS):

```
void LcpContext::sendConfigReq()
{
    _trasnmitter->sendConfigReq(this);
}
```

The declaration of the base class LcpState is shown below. Again, we have declared methods for each of the events listed in the PPP LCP specification in RFC-1661.

```
class LcpState
{
public:
      virtual ~LcpState();
      virtual void up(LcpContext* lcpContext);
      virtual void down(LcpContext* lcpContext);
      virtual void open(LcpContext* lcpContext);
      virtual void close(LcpContext* lcpContext);
      virtual void timeOut(LcpContext* lcpContext);
      virtual void recvConfigReq(LcpContext* lcpContext);
      virtual void recvConfigAck(LcpContext* lcpContext);
      virtual void recvConfigNak(LcpContext* lcpContext);
      virtual void recvTermReq(LcpContext* lcpContext);
      virtual void recvTermAck(LcpContext* lcpContext);
      virtual void recvUnknownCode(LcpContext* lcpContext);
      virtual void recvCodeRej(LcpContext* lcpContext);
      virtual void recvProtRej(LcpContext* lcpContext);
      virtual void recvEchoReq(LcpContext* lcpContext);
      virtual void recvEchoReply(LcpContext* lcpContext);
      virtual void recvDiscardReq(LcpContext* lcpContext);

protected:
      void changeLcpState(LcpContext* lcpContext, LcpState* state);
};
```

Each concrete state class will override the default event methods defined by the LcpState base class except when a specific event is invalid for that state. These cases are shown indicated by blank entries in the state table. In these cases, we simply log the error.

```
LcpState::up(LcpContext* lcpContext()
{
    Logger::log("Invalid LCP Event up Receved");
}
```

The Concrete LcpState Classes

We now go on to define concrete LcpState classes for each of the ten states specified in RFC-161. In each of the concrete LcpState classes we simply define methods to handle each of the valid events that can be received in that state. These methods simply execute the actions defined in the PPP LCP State Table. For example, here is the rcvConfigAck method for the AckSentLcpState class:

```
AckSentLcpState::rcvConfigAck(LcpContext* lcpContext)
{
    lcpContext->initRetryCount();
    lcpContext->thisLayerUp();
    changeLcpState(lcpContext, getInstance::OpenedLcpState());
}
```

Execution Efficiency

Now we will look at some coding techniques for execution efficiency. Many C programmers are concerned with the overhead associated with using C++ virtual functions and inheritance so I will address those concerns first. Then I will discuss some design level strategies for efficient coding in C++; minimizing context switching, lockless queues, and avoiding dynamic memory allocation..
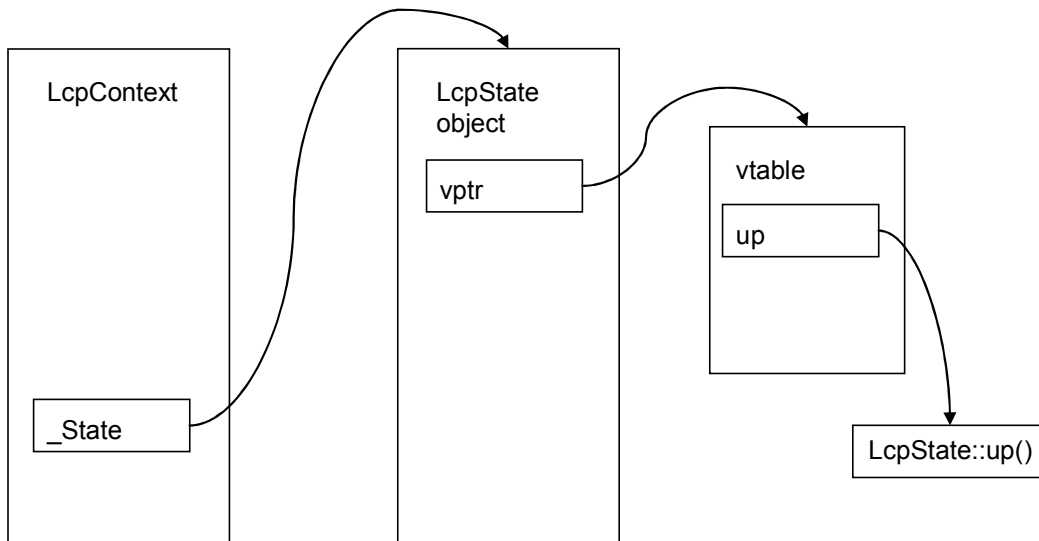
Virtual Function Overhead

Polymorphism using virtual methods and inheritance are the hall marks of object oriented programming in C++ so it is important to examine the run time costs of using these language features. The code generated by different compilers can vary; most C++ compilers use the following strategy for implementing virtual functions.

Let's go back to our LcpState and AckSentLcpState classes. The compiler must implement a way to determine which version of the event methods to execute based on the type of the LcpState object pointer, _state, in the LcpContext. This is usually done by the C++ compiler by adding a table of function pointers to the class. This table is refered to as the virtual table. If we consider only the first three event methods in out LcpState classes the virtual table might look like this:

LcpState		AckSentLcpState
LcpState::up()		LcpState::up()
LcpState::down()		AckSentLcpState::down()
LcpState::open()		AckSentLcpState::open()

Since the AckSentLcpState class does not implement an up method, the up methods slot in the table is filled in with the default up method defined in the LcpState base class.

How is the virtual table found at run time? Each object that has virtual functions contains a hidden data member, the virtual pointer, placed in it by the compiler. At run time, _state points to an LcpState object and the object's virtual pointer points to the virtual table. The equivalent C code looks something like this:

```
LcpContext._state->vptr->(void vtable[UP])();
```

So the added cost is the time it takes to resolve two pointers. The other cost is in the memory required to hold the virtual table. For our state machine the table is quite large but there usually not many states, less than a dozen for most protocols.

Minimize Context Switching

Context switching can incur a huge execution cost. First there is the cost of saving and restoring the current context. The cost is even greater on a high performance processor where a context switch negates the effectiveness of instruction prefetching, chaching, and pipelining.

Context switching can be minimized by carefully designing the threading or tasking for the protocol stack. Traditionally protocol stacks have been designed with separate threads for separate layers. Van Jacobson, a well known network researcher and author of several RFCs once said: "Layers are a good way to think about communications protocols but a lousy way to implement them." I would change this statement slightly. Translating a human readable protocol specification into a programming language that is both readable by a computer or a human requires a lot of thinking. We use languages like C++ because they allow us to use higher levels of abstraction that more closely represent the real world concepts we are trying to implement. Since communications protocols are specified in layers we should use the power of C++ to model this concept in our code. Layers are a good way to code communications protocols but a lousy way to execute that code.

Different threads are really only required when there is a speed mismatch between physical processes so that software servicing those processes must be executed asynchronously. Using TFTP to download files to a FLASH file system for example might only require three threads, one to process received packets, one to write data to FLASH, and one thread to transmit acknowledgement packets.
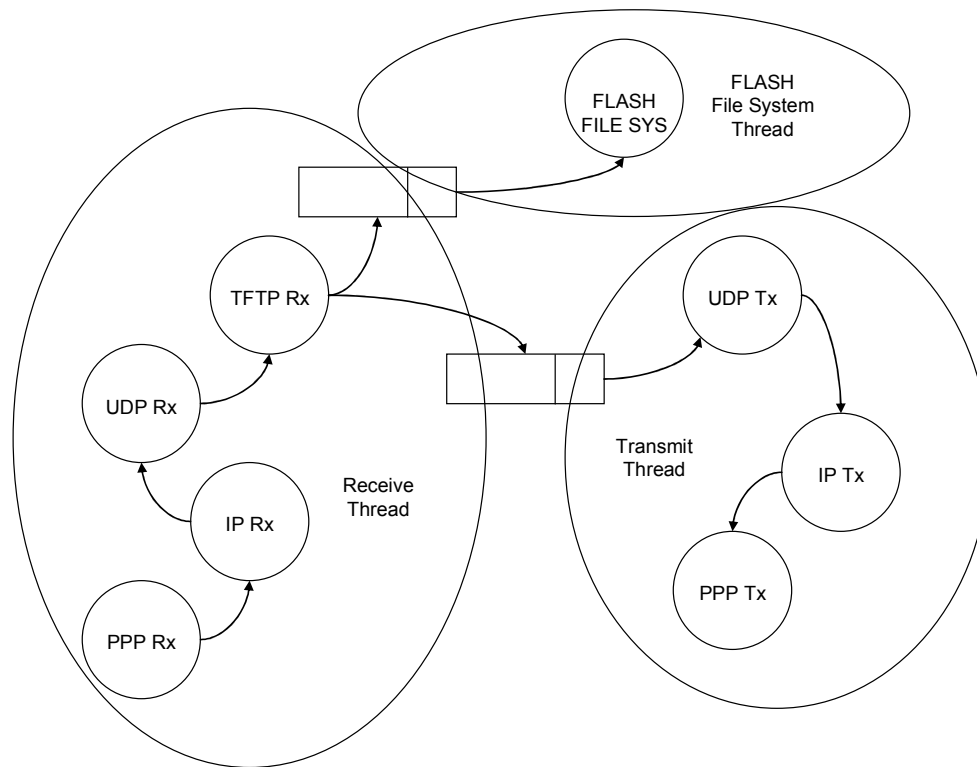
Lockless Queue

Another source of context switches is system calls. Every time a system call is made, there are two context switches. First when the operating system is invokes, second, when the operating system passes control to the highest priority run-able task. Minimizing the number of tasks will not help if every time a message is placed in a queue, the operating system is invoked. This can be avoided by using lockless queues.

A lockless queue is a queue that is written to be thread-safe without using operating system synchronization functions. This is possible when there is only one thread writing to the queue and only one thread reading from the queue.

Queue Empty

Items in Queue

Queue Full
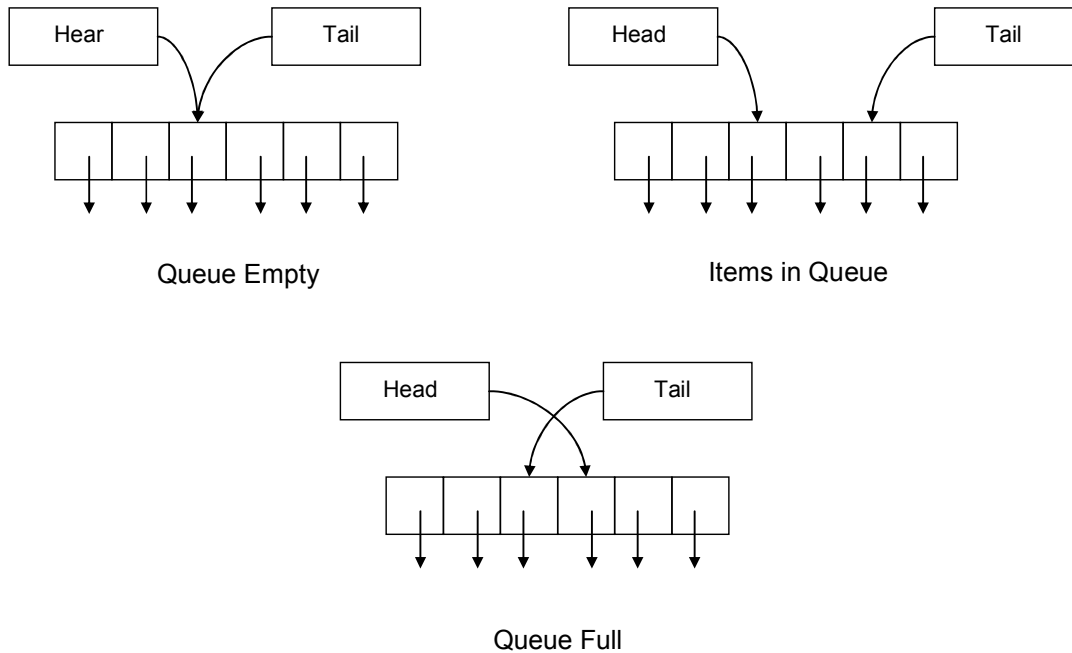
The lockless queue consists of an array of object pointers, a head pointer and a tail pointer. The key to the lockless queue is that the writing thread is only allowed to modify the head pointer and the reading thread is only allowed to modify the tail pointer.

Here is the constructor. The queue itself is an array of pointers to unsigned char i.e. octets or bytes. The head and tail are indexes into this array. I want the queue size to be a power of 2 so that I can do some optimization when the array indexes need to roll over from the end of the array back to zero.

```
// Size is specified as power of 2
// i.e. if size = n then queue size is 1 << n

LocklessQueue::LocklessQueue(unsigned int size) :
    _size(1 << size),
    _mask(_size - 1),
    _head(0),
    _tail(0)
{
    _queue = new unsigned char*[_size];
}
```

Next is the push method which pushes a new message buffer into the queue. We make a local copy of the _head index to work with. Note that while push reads the tail index, it only modifies the head index. In the worst case another thread is doing a pop operation while push is being executed. If push reads a stale version of tail the queue might appear full when it has one entry left, so the buffer would be discarded.

```
void LocklessQueue::push(unsigned char *buffer, bool *full)
{
    unsigned int tail = _tail;

    if( (tail++) & _mask == _head)
    {
        *full = true;
    }
    else
    {
        _queue[tail] = buffer;
        _tail = tail;
        *full = false;
    }
}
```

Note the head index is masked with the _mask value that was calculated in the constructor from the size. This does the required roll over to zero at the end of the array with about the same overhead as a compare to queue size that would be required otherwise.

The pop method is shown next. The pop method reads the head index but only modifies tail. The worst case in the pop method is if a stale _head value is read. The queue would appear empty when there is one message in the queue. The thread calling the pop method would simply miss an opportunity to process the message. When the queue is almost empty this should not pose a performance problem.

```
unsigned char *LocklessQueue::pop(bool *empty)
{
    unsigned char *buffer = 0;
    unsigned int head = _head;

    if(_head != tail)
    {
        buffer = _queue[head];
        head = (head + 1) & _mask;
        _head = head;
        *empty = false;
    }
    else
    {
        *empty = true;
    }

    return buffer;
}
```

Note that this function requires that the steps be done in the order they are coded. It may be necessary to turn off optimization for the lockless queue class though I haven't experienced any problems with this implementation using optimization with several compilers on different processors.

There are more complex lockless data structures that can be applied to inter-process communication or dynamic memory allocation. These data structures are implemented using special atomic processor instructions so they must be partially coded in assembly language.

Avoid Dynamically Allocating and Freeing Memory

C++ is notorious for unexpectedly creating copies of objects but this reputation is not deserved. Most of the time you want to pass pointers to objects or references to objects but if you are not careful about this C++ will make temporary copies. This can happen when you pass an object as a parameter to a function or return an object from a function. You can prevent accidentally passing objects around when you want to pass pointers or references by making the copy constructor and the = operator private.

```
Class MyClass
{
public:
    ------
private:

    MyClass(const MyClass&);
    MyClass& operator=(const MyClass&);
};
```

If you define a class this way and try to pass a copy of an object instead of a pointer, the compiler will complain about it.

Alternative new() and delete()

Some types of objects such as message buffers are created and deleted over-and-over in networking software. The default implementation for new() and delete() is to use malloc() and free() from the standard C library for managing memory. In cases where you are using the same types of objects over-and-over an optimized memory manager can be used for these specific objects. new() and delete() are simply operators that you can over-ride like any other operator in C++.

There are several alternatives for obtaining and managing memory for your more dynamic objects. A pool of objects can be defined statically if the system's resource needs are well understood. malloc() can be used to create a large block of memory for storing a number of objects at a time so that malloc() and free() aren't called as often. Another alternative is to cache objects once the have been allocated using malloc(). The objects are never really freed but instead are saved in a pool for reuse. When the pool becomes empty, additional objects are created in the usual way using malloc(). A variation on this approach is the cache a specific number of objects, freeing addition objects if the pool is full and allocating new ones when the pool becomes empty.

Conclusion

Using object oriented programming in C++ can improve the developer's efficiency in implementing communications protocols. We have seen one example of how using the State pattern simplifies developing the code for the PPP Link Control Protocol. With careful use, C++ does not add intolerable overhead to executing communications protocol software. Several strategies were presented for developing efficient communications protocol software at a system level demonstrating that C++ can be used to develop high performance networking software.

References

Simpson, W. *The Point-to-Point Protocol(PPP) RFC-1661.* Daydreamer Computer Systems Consulting Services, July 1994.

Gamma, E. Helm, R. Johnson, R. and Vlissides, J. *Design Patterns; Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Reading, MA 1994.

Lippman, S. *Inside the C++ Object Model.* Addison-Wesley Publishing Company, Reading, MA 1996

Meyers, S. *Effective C++; 55 Specific Ways to Improve Your Programs and Designs, Third Edition. .* Addison-Wesley Publishing Company, Reading, MA 2005.

Meyers, S. More *Effective C++; 35New Ways to Improve Your Programs and Designs, Third Edition. .* Addison-Wesley Publishing Company, Reading, MA 1996.

Jacobson, V. and Felderman, B. *Speeding Up Networking.* Linux.conf.au 2006, Dunedin, NZ

Dokumentov, A. *Lock-free Interprocess Communication.* Dr. Dobbs Portal, June 15, 2006.

Harvey Sugar is a senior software engineer at Mantaro Product Development Services, Inc. He has over 25 years of experience in systems and software engineering, developing real-time embedded systems primarily for telecommunications and data network products. He has worked with communications technologies including; packet switching, optical networking, satellite communications, telephony, and wireless. He has implemented communications protocols including TCP/IP, Frame Relay, ATM, SS7 and Packet Over SONET. Some of his accomplishments include: pioneering the use of Object Oriented design and C++ programming in developing a software framework for telecommunications test equipment and leading the development of the first production ADSL test set Harvey can be reachd via email at hsugar@mantaro.com.