Asynchrony for the Masses

Marc Duerner

Multi-Threading is hard.

If we are not careful we get stuck, because of deadlocks, livelocks or uncooperative threads that do not respond anymore. It is shown here how to avoid these problems with a portable and productive API, that also delivers high performance, determinism and reliability.

Threading for the Masses

- How do I stop a blocked Thread?
  - We know the solution is **not** to terminate or kill the thread:
    - Interrupt Politely, Dr. Dobbs Journal, April 2008, Herb Sutter, http://drdobbs.com/cpp/207100682
  - This is an unsolved problem so far, the only answer to this is simply not to block
  - Blocking is a sign of a deadly disease and the OS might simply put an end to your suffering (watchdog)

- How many threads should I start?
  - For performance reasons: One thread per CPU
  - Having more threads will not make your app scale better, because of context switching
  - Intel TBB lets you tune the number of threads in the pool

- But: The world isnt‘t perfect
  - We must not block the GUI main loop so we place long tasks in a worker thread
  - We only do this because we have no other choice
  - Rule 1: never block the UI thread (ok, we knew this already)
  - Rule 2: never block a server thread (ok, I can see why…)
  - Rule 3: never block a worker thread (WTF ???)

When Timers go wrong

```
void thread_function()
{
    while(keepRunning)
    {
        sleep(interval);
        observer->notify();
    }
}
```

- The code above will make experienced developers cry (…or laugh)
  - Timer threads are a great example of completely unneccessary threading
  - Can you spot all the other bugs?

Timers Done Right (1)

```
void onTimeout()
{
    std::cout << „timeout“ << std::endl;
}

int main(int argc, char** argv)
{
    Pt::System::MainLoop loop;

    Pt::System::Timer timer;
    timer.setActive(loop);
    timer.timeout() += Pt::slot(&onTimeout);
    timer.timeout() += Pt::slot(loop, &MainLoop::exit);
    timer.start(5000);

    loop.run();
    return 0;
}
```

Timers Done Right (2)

- Program blocks on run() with 0% CPU time if nothing happens
  - Please, no burning loops
  - Save battery (and the environment)

- If something happens, we are notified by signals and delegates
  - Safe, fast and flexible callback mechanism (cost: twice as much as virtual call)
  - Signal is 1:N relationship, Delegate is 1:1 relationship
  - Used for notification within a thread context
  - Automatic disconnect when slot or signal/delegate are destructed

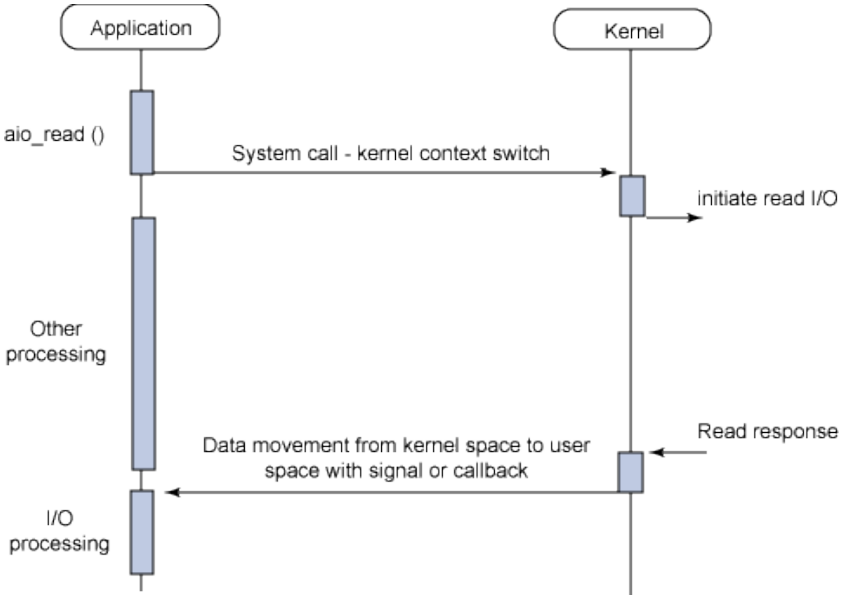
- Timer monitored by an event loop
  - Deregisters automatically when loop or timer are destoyed
  - Easy life-time management
  - Notifies through timeout signal

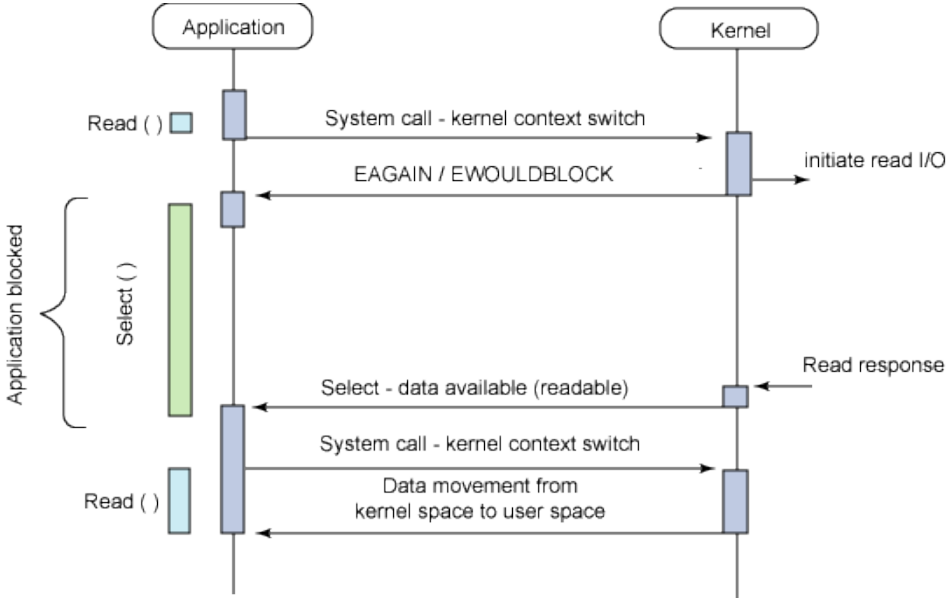
- Interval based, drift-free
  - Unless you block the event loop for longer than the timer inteval
  - Accuracy depends on OS

- Cheap
  - Requires 64 bytes when registered in an event loop
  - Event loop is not slowed down when many timers are registered

Proactors vs Reactors

- Proactor (asynchronous non-blocking)
  - Long tasks run asynchronously
  - Completion handler called (or some other kind of notification)
  - Must be cancellable
  - Example: POSIX AIO (hint: buffer needed at start of operation)



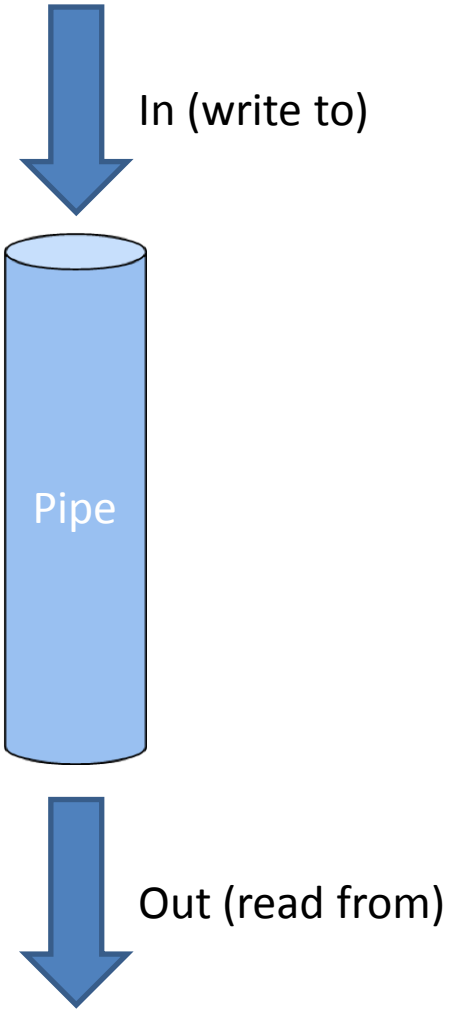


- Reactor (asynchronous blocking)
  - Block on all resources
  - Notifies when operation can be performed without blocking
  - Must be cancellable
  - Example: select/poll (hint: buffer is used later)

- OS might offer just one or both
  - Application API must support both

Asynchronous I/O (1)

```
char buffer[25];

void onWrite(Pt::System::IODevice& dev)
{
    size_t n = dev.endWrite();
}

void onRead(Pt::System::IODevice& dev)
{
    size_t n = dev.endRead();
    std::cout.write(buffer, n);
}

int main(int argc, char** argv)
{
    Pt::System::MainLoop loop;
    Pt::System::Pipe pipe;

    pipe.in().setActive(loop);
    pipe.in().beginWrite("Hello World", 11);
    pipe.in().outputReady() += Pt::slot(&onWrite);

    pipe.out().setActive(loop);
    pipe.out().beginRead( buffer, sizeof(buffer) );
    pipe.out().inputReady() += Pt::slot(&onRead);

    loop.run();
    return 0;
}
```



Asynchronous I/O (2)

- IODevices wrap native I/O handle
  - File or socket descriptor
  - Overlapped I/O event handles
  - Active objects
  - IODevices for Files, Pipes, SerialDevices, TCP, UDP, parent/child process I/O
- IODevice monitored by an event loop
  - Deregisters when loop or IODevice are destroyed
  - Notifies through inputReady and outputReady signals
  - Begin-notify-end pattern
- Solves C10k problem
  - http://www.kegel.com/c10k.html
  - „It's time for web servers to handle ten thousand clients simultaneously, don't you think? After all, the web is a big place now. “
- Renaissance of reactive I/O
  - In the olden days (before MT) it used to be your only choice !!!
  - 90‘s were the dark ages of multi-threading (Hart J.M., Windows System Programming)
  - Some OS don‘t offer blocking API anymore (Symbian, WinRT)
  - Newest generation of web servers work this way: nginx, Cherokee
  - Blocking I/O is vulnerable to attacks such as Slowloris DOS
  - epoll since Linux 2.5.44
  - Kqueue/knotify since FreeBSD 4.1

Asynchronous I/O with streams (1)

```
void onWrite(Pt::System::IOBuffer& sb)
{
    size_t n = sb.endWrite();

    std::cout << „bytes left: “ << sb.out_avail() << std::endl;

    if(sb.out_avail() > 0)
        sb.beginWrite();
}


Pt::System::MainLoop loop;
Pt::System::Pipe pipe;

pipe.in().setActive(loop);
Pt::System::IOBuffer sb(pipe.in());
sb.outputReady() += Pt::slot(&onWrite);

std::ostream os(&sb);
os << „Hello world!“ << std::endl;
sb.beginWrite();

loop.run();
```

Asynchronous I/O with streams (2)

```
void onRead(Pt::System::IOBuffer& sb)
{
    size_t n = sb.endRead();

    std::cout << „bytes available: “ << sb.in_avail() << std::endl;

    char in[20];
    std::istream is(&sb);
    is.readsome(in, sizeof(in));
}


Pt::System::MainLoop loop;
Pt::System::Pipe pipe;

pipe.out().setActive(loop);
Pt::System::IOBuffer sb(pipe.out());
sb.inputReady() += Pt::slot(&onRead);
sb.beginRead();

loop.run();
```

Asynchronous I/O with streams (3)

- With streams we mean std::iostreams with std::streambufs
  - Again, begin-notify-end
  - IODevice works on std::streambuf‘s buffer areas
  - IOBuffer can work with any IODevice

- Output buffer:
  - Enlarge buffer area on overflow
  - We don‘t know up-front how much data we want to write
  - Possible strategy: Start writing when buffer exceeds a water mark
  - Query buffer size with out_avail()
  - On successful write only part of buffer might be written out
  - Important to optimize write() system call

- Input Buffer:
  - On successful read only part might be filled
  - Optimize read system call
  - Check buffer size with in_avail();

Asynchronous I/O - Error Handling (1)

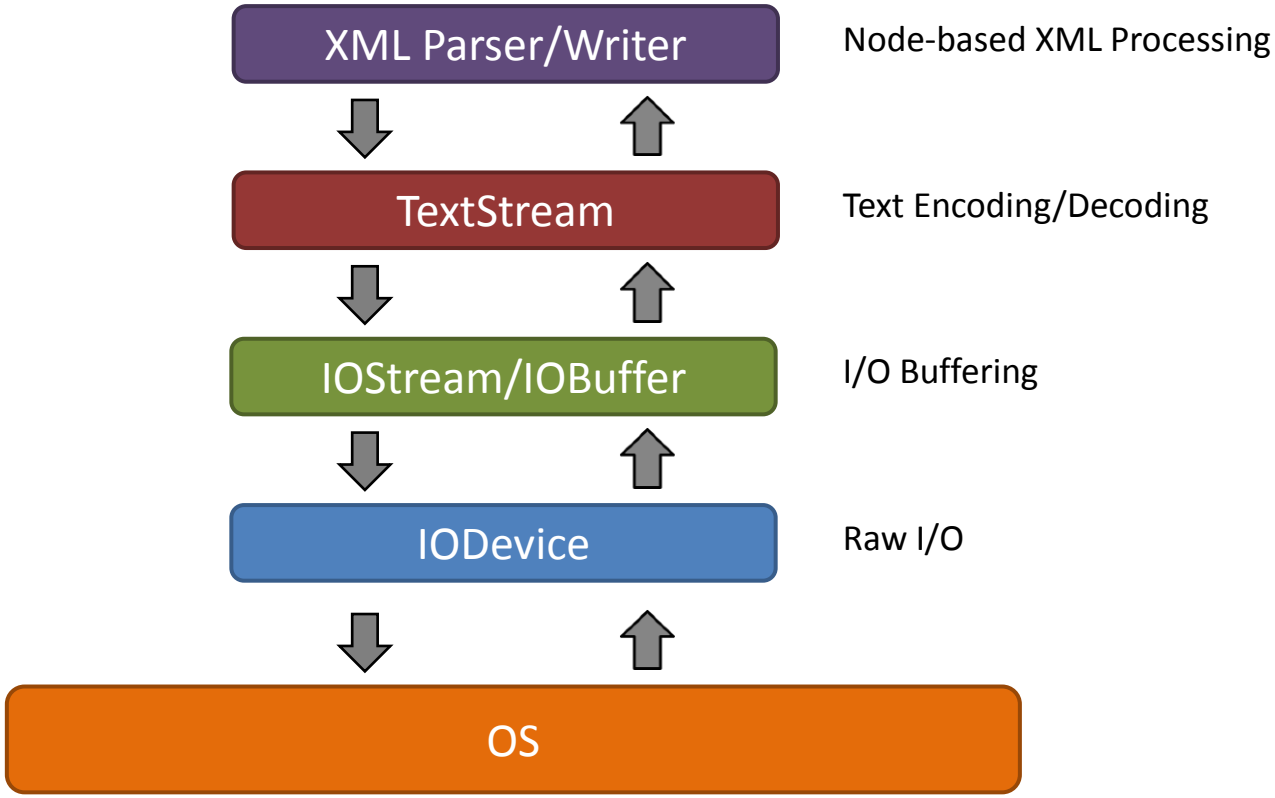
```
void onRead(Pt::System::IOBuffer& sb)
{
    try
    {
        size_t n = sb.endRead(); // throws

        char in[20];
        std::istream is(&sb);
        is.readsome(in, sizeof(in));
    }
    catch(const Pt::System::IOError& e)
    {
        // handle I/O error, close device etc…
    }
}
```

Asynchronous I/O – Error Handling (2)

- All exceptions are defered until the operation is ended
  - IODevice::endRead
  - IODevice::endWrite
  - IOBuffer::endRead
  - IOBuffer::endWrite
  - We will see more later, fits well with begin-notify-end pattern
  - If exceptions were just thrown where they appear we would be kicked out of the loop
  - Need to deal with exceptions in the callback

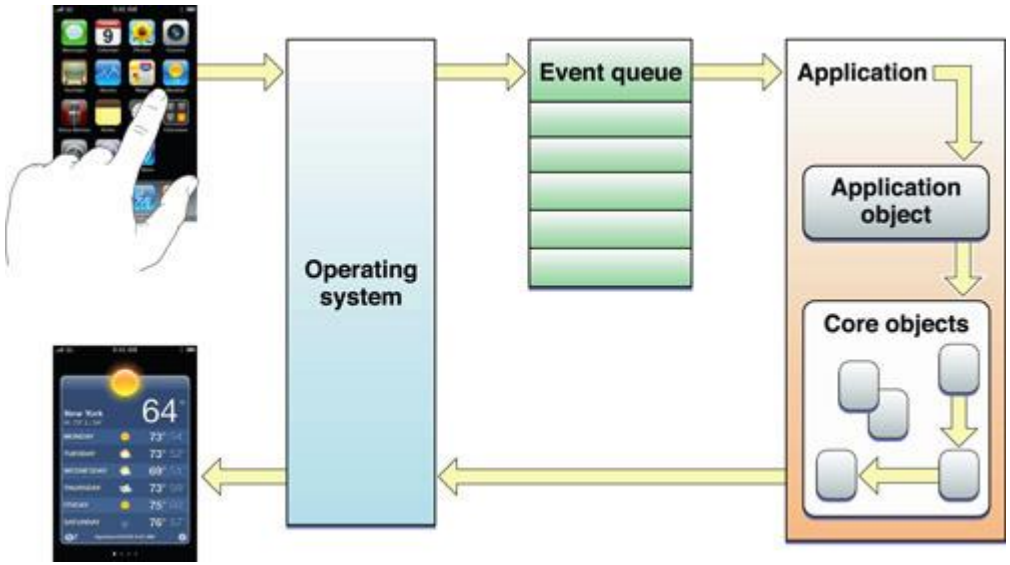
- IOErrors can be cured by closing the connection / device

Asynchronous I/O Software Stack



Event Driven Programming

- Not a new technology
  - Traditionally, all UI are event-driven
  - Events are stored by the OS in the event queue of an application
  - Application wakes up and processes events in ist context
  - Events are routed to receivers (publish-subscribe)



⇨ Can be leveraged for inter-thread communication

Inter-Thread Event Handling

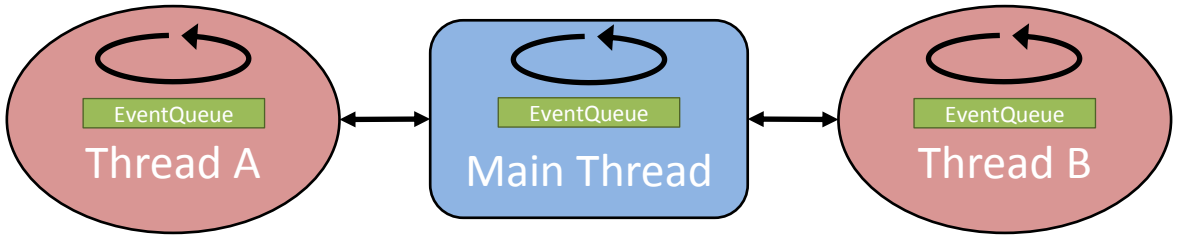
```
class ExitEvent : public Pt::Event {};

void threadFunc()
{
    ExitEvent ev;
    Pt::System::Application::instance().loop().commitEvent(ev);
}

void onExitEvent(const ExitEvent& ev)
{
    Pt::System::Application::instance().loop().exit();
}

int main(int argc, char** argv)
{
    Pt::System::Application app(argc, argv);
    app.loop().eventReceived() += Pt::slot(&onExitEvent);

    Pt::System::AttachedThread thread(Pt::callable(&threadFunc));
    thread.start();

    app.run();
    return 0;
}
```

Inter-Thread Event Handling

- Threads communicate through the means of events and a thread-safe loop API
  - Use events to cross thread borders
  - commitEvent() puts event into queue and wakes up loop
  - queueEvent() only puts event into queue
  - wake() forces a loop iteration
  - exit() quits an event loop

- Signal dispatches events within a thread
  - No manual cast required.
  - Event type derived from signature: `app.loop().eventReceived() += Pt::slot(&onExitEvent);`

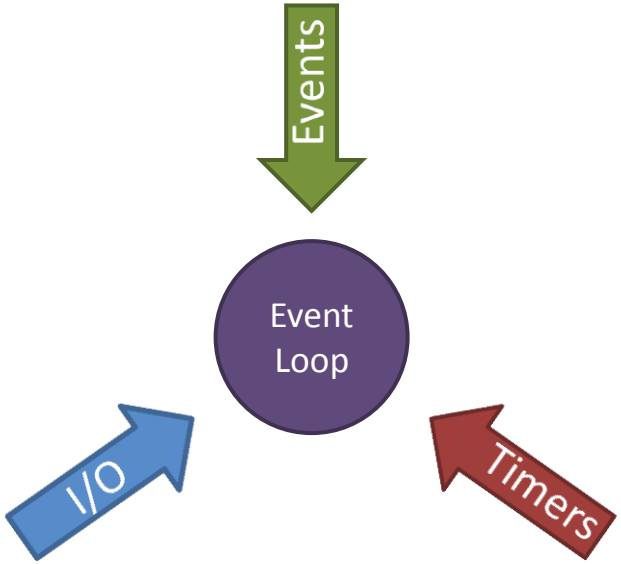
- Ideal: Per-Thread event loop
  - Performance: one active thread per CPU
  - Performance: much better locality of reference
  - Performance: within a thread we don‘t need to lock (example: allocators)



The Big Picture

- Combination of asynchronous I/O, Events and Timers
  - Reliably stoppable service threads
  - Reduce total number of required threads
  - Less locking
  - Clear separation of responsibility

One ring to rule them all, one ring to find them,
One ring to bring them all and in the darkness bind them



Implementation Hints

```
DWORD WaitForMultipleObjects(DWORD nCount,
                             const HANDLE* lpHandles,
                             BOOL bWaitAll,
                             DWORD dwMilliseconds );
```

- Register I/O handles (IODevice) as fd_set (non-blocking) or HANDLE (overlapped)
- Begin-notify-end can handle level and edge triggering
- Register windows event or pipe handle to wake up if events need to be processed
- Suspend timeout is interval to next timer in timer queue

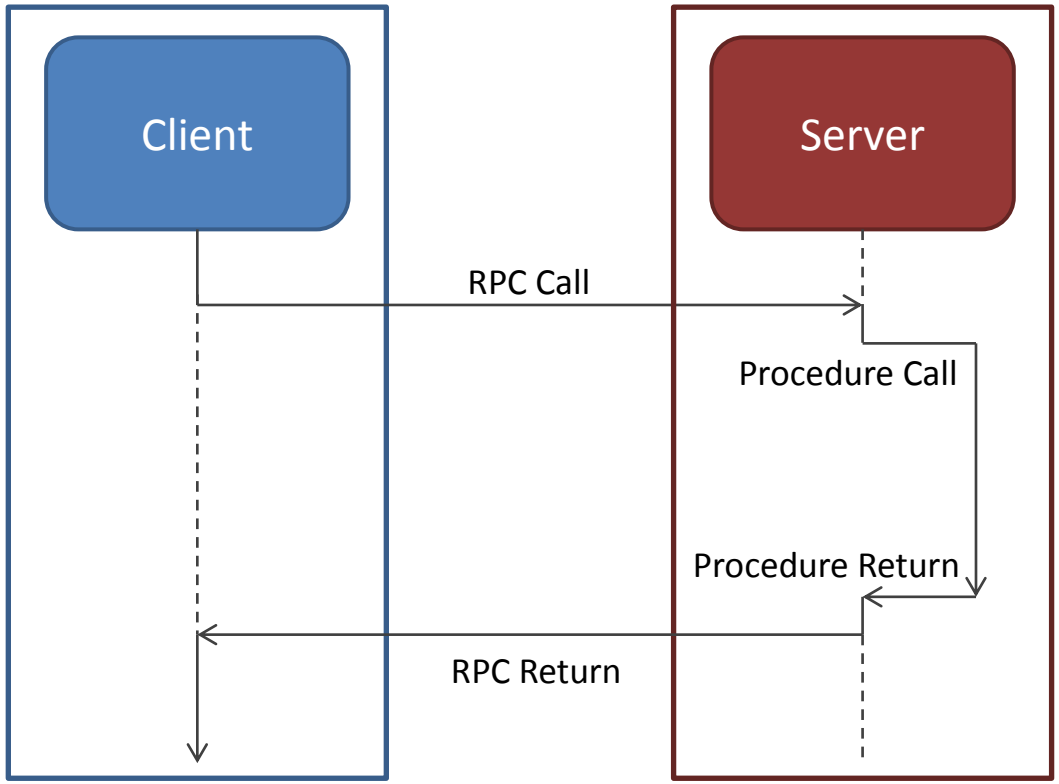
```
int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

Works also for:

- NSRunLoop / CFRunLoop
- QNX Photon
- Epoll
- Poll
- Kqueue/knotify
- POSIX AIO
- Symbian ActiveObjects/ActiveScheduler

Remote Procedure Calls

- Client Server Communication
  - Client serializes arguments and sends data to server
  - Server deserializes arguments and calls local procedure
  - Server serializes return value and sends data to client
  - Client deserializes return value
- Key issues are network I/O and serialization



Reliable XML-RPC Clients

```
void onMultiply(const Pt::XmlRpc::Result<int>& result)
{
    try
    {
        int r = result.get();
        std::cout << „result: “ << r << std::endl;
    }
    catch(const Pt::XmlRpc::Fault& fault)
    {
        std::cerr << fault.what() << std::endl;
    }
    catch(const Pt::System::IOError& e)
    {
        std::cerr << e.what() << std::endl;
    }
}

int main(int argc, char** argv)
{
    Pt::System::MainLoop loop;

    Pt::Net::Endpoint ep("127.0.0.1", 4000);
    Pt::XmlRpc::HttpClient client(loop, ep, "/calc");

    Pt::XmlRpc::RemoteProcedure<int, int, int> multiply(client, „multiply“);
    multiply.finished() += Pt::slot(&onMultiply);
    multiply.begin(6, 7);

    loop.run();
    return 0;
}
```

Reliable XML-RPC Services

```
int multiply(int a, int b)
{
    if( ckeckOverrun(a, b) )
        throw Pt::XmlRpc::Fault(„integer overrun“, 40);

    return a * b;
}

int add(int a, int b)
{
    if( ckeckOverrun(a, b) )
        throw Pt::XmlRpc::Fault(„integer overrun“, 40);

    return a + b;
}

int main(int argc, char** argv)
{
    Pt::System::MainLoop loop;

    Pt::XmlRpc::ServiceDefinition def;
    service.registerProcedure(„multiply“, &multiply);
    service.registerProcedure(„add“, &add);

    Pt::XmlRpc::HttpService service(def);

    Pt::Http::Server server(loop, "127.0.0.1", 4000);

    Pt::Http::MapUrl mapUrl(„/calc“, service);
    server.addServlet(mapUrl);

    loop.run();
    return 0;
}
```

Reliable XML-RPC Clients and Services (1)

- Maps to function or method call
  - Almost like calling a local function
  - Begin-notify-end pattern with function object
  - Can export any class as a service
  - Can register normal existing functions as service prodecures
  - We can throw on service side and catch on client side !!!
- Fully non-blocking API available
  - Many remote calls from one thread
  - No need for synchronisation
  - No need for extra thread, mix with timers, events and other I/O
  - Cancel and immediate stop possible
- Does not need WSDL or IDL etc…
  - Wait a minute! What about complex objects as parameters?

Reliable XML-RPC Clients and Services (2)

- But I want this:

```
Pt::XmlRpc::RemoteProcedure<PositionInfo> getPosition(„getPosition“);
```

- Does not compile until framwork knows how to serialize PositionInfo
  - Only need to implement two operators

- Not Good: serializers as generated code according to some description language
  - Complicates build process, need to edit generated code
  - Generated code is often limited in various ways
  - Locked into an alien type system

- What Pt does instead: generic serialization to support **any** kind of RPC
  - Compile time saftety
  - Requires only C++ language features

Reliable XML-RPC Clients and Services (3)

```
class PositionInfo
{
  friend void operator>>=(const Pt::SerializationInfo& si, PositionInfo& p);
  friend void operator<<=(Pt::SerializationInfo& si, const PositionInfo& p);

  public:
    PositionInfo()
    : _x(0), _y(0)
    {}

  private:
    Pt::uint32_t _x;
    Pt::uint32_t _y;
};

void operator>>=(const Pt::SerializationInfo& si, PositionInfo& p)
{
  si.getMember(„x") >>= p._x;
  si.getMember(„y") >>= p._y;
}

void operator<<=(Pt::SerializationInfo& si, const PositionInfo& p)
{
  si.setTypeName(„PositionInfo");
  si.addMember(„x") <<= p._x;
  si.addMember(„y") <<= p._y;
}
```

Reliable XML-RPC Clients and Services (4)

- Serialization is non-intrusive
  - Don‘t need to derive some interface
- Use your types
  - std::string, CString, WxString **all work in the same way**, none is preferred
  - No needless conversions from alien string, array etc…
- Object graph can be build up and broken-down incrementally
  - Prerequisite for I/O multiplexing
  - Also true for parser, text encoder etc…
- It gets even better: custom composer and decomposer
  - Use less memory
  - Higher performance
  - Handle really large objects/collections
  - More difficult to implement