

---

Subject: Re: Kqueue/epoll based interface for TcpSocket and WebSocket

Posted by [mirek](#) on Mon, 30 Apr 2018 09:20:50 GMT

[View Forum Message](#) <> [Reply to Message](#)

---

Sorry for the delay...

Quote:

```
template <class T_SOCKET>
class SocketEventQueue: NoCopy
{
public:
    SocketEventQueue(): errorCode(NOERR) { InitEventQueue(); }
    ~SocketEventQueue();

    bool ClearEventQueue();
    QueueHandler GetQueueHandler() const;

    bool IsError() const { return errorCode != NOERR; }
    ErrorCode GetErrorCode();

    bool SubscribeSocketRead(const T_SOCKET &sock);
    bool SubscribeSocketWrite(const T_SOCKET &sock);
    bool SubscribeSocketReadWrite(const T_SOCKET &sock);

    bool DisableSocketRead(const T_SOCKET &sock);
    bool DisableSocketWrite(const T_SOCKET &sock);
    bool DisableSocketReadWrite(const T_SOCKET &sock);

    bool RemoveSocket(const T_SOCKET &sock);

    bool IsSocketSubscribedRead(const T_SOCKET &sock) const { return IsSocketSubscribed(sock,
WAIT_READ); }
    bool IsSocketSubscribedWrite(const T_SOCKET &sock) const { return IsSocketSubscribed(sock,
WAIT_WRITE); }

    bool IsSocketDisabledRead(const T_SOCKET &sock) const { return IsSocketDisabled(sock,
WAIT_READ); }
    bool IsSocketDisabledWrite(const T_SOCKET &sock) const { return IsSocketDisabled(sock,
WAIT_WRITE); }

    Vector<SocketEvent<T_SOCKET>> Wait(int timeout);
};
```

Why is T\_SOCKET a template parameter? Because of websocket?

What is DisableSocketRead supposed to do? Opposite of Subscribe?

If I am right about T\_SOCKET, I have to disagree with the interface a bit.

Particularly, I think

```
Vector<SocketEvent<T_SOCKET>> Wait(int timeout);
```

is clumsy - this will IMO cause problems with mapping T\_SOCKET back to its "processes".

When I was thinking about how to proceed with this, I was considering to simply expand SocketWaitEvent. I believe that the best would probably be to treat it as full array of sockets, using indices to identify the 'process'. Something like

```
class SocketWaitEvent {
.....
public:
void Clear() { socket.Clear(); }
void Add(SOCKET s, dword events) { socket.Add(MakeTuple((int)s, events)); }
void Add(TcpSocket& s, dword events) { Add(s.GetSOCKET(), events); }

int Wait(int timeout);
dword Get(int i) const;
dword operator[](int i) const { return Get(i); }

// new:
void Set(int ii, TcpSocket& s, dword events);
void Insert(int ii, TcpSocket& s, dword events);
void Remove(int ii, TcpSocket& s, dowrd events);

Vector<int> WaitEvent(int timeout);
// or perhaps
Vector<Tuple<int, dword>> WaitEvent(int timeout); // dword part contains Get bitmask

// maybe:
void Clear(int ii); // makes index empty
int FindEmpty() const; // finds the first index that is empty

SocketWaitEvent();
};
```

Is there a reason to make things more complicated than this?

Quote:

The first problem is related to the current implementation of TcpSocket::RawWait(...). It uses

select(...) system call for determining possibility of reading/writing data or exceptional state of socket. As far as I can understand, it means that server with large number of sockets will work slowly anyway. I patched TcpSocket::RawWait(...) for BSD platform on my local machine (see below) before I started to implement the interface. Now I think that it's possible to use SocketEventQueue in purpose of determining socket state instead of raw kqueue/epoll/select. What do you think about this?

I agree.

Quote:

But there is another problem related to kqueue/epoll reaction on socket closing for both my patch and SocketEventQueue. So here's the patch:

```
#ifdef PLATFORM_BSD
timespec *tvalp = NULL;
timespec tval;
if(end_time != INT_MAX || WhenWait) {
    to = max(to, 0);
    tval.tv_sec = to / 1000;
    tval.tv_nsec = 1000000 * (to % 1000);
    tvalp = &tval;
    if (to)
        LLOG("RawWait timeout: " << to);
}

struct kevent eventrx, eventw;
struct kevent triggeredEvents[2];
int kq;
int eventFlags = EV_ADD | EV_ONESHOT;

if( ( kq = kqueue() ) == -1 ) // queue fd should be created once at the moment of socket opening
{
    // and closed at the moment of socket closing
    // the same is for SocketEventQueue object

    LLOG("kq = kqueue() returned -1");
    SetSockError("wait");
    return false;
}

if(flags & WAIT_READ)
{
    EV_SET( &eventrx, socket, EVFILT_READ, eventFlags, 0, 0, NULL );
    if( kevent( kq, &eventrx, 1, NULL, 0, NULL ) == -1 )
    {
        LLOG("kevent( kq, &eventrx, 1, NULL, 0, NULL ) returned -1");
    }
}
```

```

SetSockError("wait");
close(kq);
return false;
}
}

```

```

if(flags & WAIT_WRITE)
{
EV_SET( &eventw, socket, EVFILT_WRITE, eventFlags, 0, 0, NULL );
if( kevent( kq, &eventw, 1, NULL, 0, NULL ) == -1 )
{
LLOG("kevent( kq, &eventw, 1, NULL, 0, NULL ) returned -1");
SetSockError("wait");
close(kq);
return false;
}
}
}

```

```

int avail = kevent( kq, nullptr, 0, triggeredEvents, 2, tvalp ); // here is the problem if
socket

```

```

// works in blocking mode
// or if timeout is too long

```

```

close( kq );

```

```

#else

```

```

// default select implementation

```

Now let's imagine the situation:

```

TcpSocket server; // passes through TcpSocket::Listen()

```

```

...

```

```

void Server() // runs in several threads

```

```

{
static StaticMutex serverMutex;

while(!Thread::IsShutdownThreads())
{
TcpSocket client;
bool acceptStatus;

{
Mutex::Lock ____(serverMutex);
//acception is in blocking mode
acceptStatus = client.Accept(server); // calls TcpSocket::RawWait(...)
}
}
}

```

```

... // connection handling
}
}

...

void SignalHandler(int sig)
{
server.Close(); // Doesn't interrupt kevent system call in TcpSocket::RawWait(...)
// close(socket) just makes kqueue to delete all events
// associated with socket descriptor from it's kernel queue

Thread::ShutdownThreads();
}

```

So I can't normally terminate the server if I work with sockets in blocking mode. Do you have any ideas how to interrupt kevent waiting loop? I've tried to call shutdown(socket, SD\_BOTH) for sockets that hadn't been passed through TcpSocket::Listen(), and it works for me. But I still can't deal with listening socket. Solution I've found is to use pipe-trick: read-end descriptor attaches to kqueue/epoll, and write-end descriptor attaches to socket. When socket closes, it writes some data in pipe with write-end descriptor. But it means that socket must hold all queue write-end descriptors it was attached. Could you help me with my problem?[/quote]

Well, I was fighting with this one too, years ago. Thats nasty little problem there.

I the end, I believe that the best solution is to make Accept return until there are any active threads by doing localhost connect.

```

TcpSocket s;
s.Connect("127.0.0.1", port);

```

You can check Skylart/App.cpp.

Another option, not always applicable, is not to bother and let the signal kill the application just as it is supposed to....