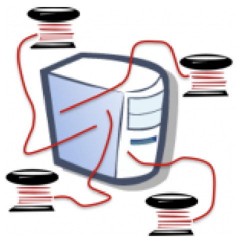
Multithreaded Work Queue Based Server in C++

Vic | March 14, 2013 | 10 comments



Creating a multithreaded TCP/IP protocol based server requires the capabilities to handle network communication, multithreading and transferring data between threads.

I have described how to build C++ components to handle this functionality in previous blogs. This time I'll show you how to combine these components to create a simple multithreaded server.

Background Articles

The core server functionality that I'll use in this project come from source code presented in the following previous articles of mine. Note you can get the source code for the articles on Github.

1. Network communication – TCP/IP Network Programming Design Patterns in C++
2. Multithreading - Java Style Thread Class in C++
3. Transferring data between threads – Multithreaded Work Queue in C++

Server Structure

Producer-Consumer Model

The server is based on the producer-consumer multithreaded model I discussed in Multithreaded Work Queue in C++, where a single producer thread passes work items to 1 or more consumer threads via a work queue, implemented with the `wqueue` class. Threads will be created using the `Thread` class discussed in Java Style Thread Class in C++.

In the case of the TCP/IP server, the producer thread accepts connections then queues the connections for the consumer threads which in turn handle the connection processing as shown in this diagram.

SEARCH

Type keywords and hit Enter

ABOUT VIC

I'm a software developer, blogger and family man enjoying life one cup of coffee at a time.

Professionally I'm a developer for the Data Analytics Group at Trend Micro and Community Manager for the OSSEC Project. I also blog for Trend Micro Simply Security and Security Intelligence.

Yeah I like coffee.

RECENT POSTS

Securing Hadoop with OSSEC

Condition Variable Class in C++

Mutex Class in C++

Makefile for HBase Applications

Switching Between Mobile and Desktop WordPress Themes

ARCHIVES BY MONTH

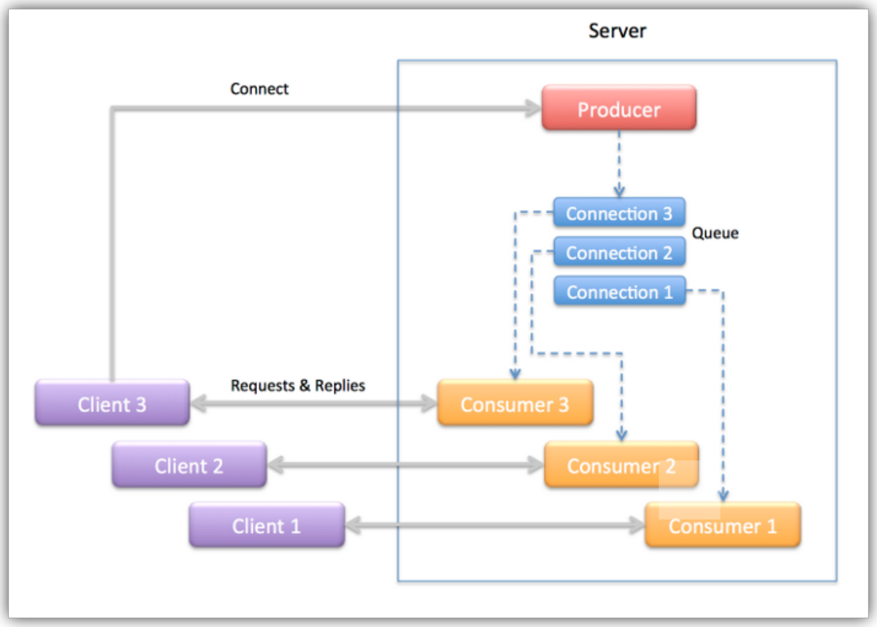
Select Month ▼

ARCHIVES BY CATEGORY

Select Category ▼

ARTICLE SUBTOPICS

Any Mobile Theme Switcher Better WP Security C/C++ Carrington CentOS CGI CSS CSV Data Structures Design Patterns DOM Hadoop HBase IntelliJ Internet Safety Java



Producer Thread

The producer thread in the server is implemented in the `main()` function. It's job is to create the work queue and consumer threads then accept connections from clients and pass the connections off to the consumer threads to handle. Specifically, the producer thread takes the following actions:

1. Create a work queue object.
2. Create the consumer threads.
3. Start listening for connections from clients.
4. Wait to accept a connections from a client using a TCPAcceptor object – discussed in the TCP/IP Network Programming Design Patterns in C++ blog.
5. For each connection create a work item that transfers the connected socket – contained in a TCPStream object – to a consumer thread to handle the connection.
6. Return to step 4.

Consumer Thread

The consumer threads are the workers that do the protocol session handling for the server. Each consumer thread handles a connection in the following manner:

1. Wait for a work item to be added to the queue.
2. Remove a work item from the queue.
3. Extract the TCPStream object from the work item.
4. Wait to receive a request from the client.
5. Process the request when it is received.
6. Send the reply back to the client.
7. Repeat steps 4 – 6 until the client closes the connection.
8. Close the server end of the connection when the client closes the connection.
9. Delete the work item.
10. Return to step 1.

Work Queue

The `wqueue` class supports the methods to add and remove work items. It encapsulates a Standard C++ `list` object along with the Pthread functions to serialize access to the work items and enable the producer thread to signal each consumer thread when items are added to the queue.

SECURITY NEWS

security
Tweets from a list by Vic Hargrave

Tripwire, Inc. @TripwireInc 1h
Stealth Pentesting: When IT Doesn't Even Know We're Here ow.ly/ql5TY via @reynoldsrb @tx3_ #BSidesDFW #infosec #security #pentest
Expand

Tripwire, Inc. @TripwireInc 1h
Security Slice: Me, You and the EU Cyber Security Directive ow.ly/qkQjL #infosec #security #EU #breaches
Expand

dragosr @dragosr 1h
I agree @donicer air gap discipline is incredibly hard. Speaking from personal DERP moment when meticulous with memstick, sharing USB CDs.
Expand

Andrew Plato @andrewplato 14h
Fascinating - Malware that can jump airgaps using ultrasonic sounds. lnkd.in/bjQNn8D
Retweeted by TRUSTIFIER INC.
Expand

CISecurity 17h

Server Application

WorkItem Class

The server code for the project resides in a single file *server.cpp*. It starts off with the headers files and the definition of the `WorkItem` class.

```
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "thread.h"
#include "wqueue.h"
#include "tcpacceptor.h"

class WorkItem
{
    TCPStream* m_stream;

  public:
    WorkItem(TCPStream* stream) : m_stream(stream) {}
    ~WorkItem() { delete m_stream; }

    TCPStream* getStream() { return m_stream; }
};
```

The constructor accepts a `TCPStream` object pointer which can be accessed through a call to the `WorkItem::getStream()` method. When the `WorkItem` object is deleted it closes the connection by deleting the `TCPStream` object.

ConnectionHandler Class – Consumer Thread

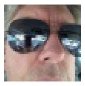
The consumer threads are implemented by the `ConnectionHandler` class which is derived from the `Thread` class. The constructor is passed a reference to the work queue created in the `main()` function.

The `run()` method implements the steps discussed in the Consumer Thread section of this article. All the thread mutex locking a condition signaling is handled internally by the work queue class so we don't have to worry about.

```
class ConnectionHandler : public Thread
{
    wqueue<WorkItem*>& m_queue;

  public:
    ConnectionHandler(wqueue<WorkItem*>& queue) : m_queue(queue) {}

    void* run() {
        // Remove 1 item at a time and process it. Blocks until an item
        // is placed on the queue.
        for (int i = 0;; i++) {
            printf("thread %lu, loop %d - waiting for item...\n",
                   (long unsigned int)self(), i);
            WorkItem* item = m_queue.remove();
            printf("thread %lu, loop %d - got one item\n",
                   (long unsigned int)self(), i);
            TCPStream* stream = item->getStream();

            // Echo messages back the client until the connection is
            // closed
            char input[256];
            int len;
            while ((len = stream->receive(input, sizeof(input)-1)) != 0 ){
                input[len] = NULL;
                stream->send(input, len);
                printf("thread %lu, echoed '%s' back to the client\n",
                       (long unsigned int)self(), input);
            }
            delete item;
        }

        // Should never get here
        return NULL;
    }
};
```

[Line 12-17] Prints the thread ID and waiting status. Blocks on the `wqueue::remove()` call until a

FACEBOOK LIKES



VicHargrave.com

Like

81 people like VicHargrave.com.

Facebook social plugin

By: Rashid Azar

BLOG DISCLAIMER

The views and opinions expressed on this blog are my own and in no way represent the views or opinions of my employer, Trend Micro.

work item is placed in the queue. Prints an indication that an item has been placed on the queue then removes the item and extracts the `TCPStream` object it contains.

[Line 23-34] Continually receives messages from the client, prints them to stdout and echoes them back to the client. When the client closes the connection, the WorkItem object is deleted then the thread returns to get another item from the queue.

Main Function – Producer Thread

The `main()` function implements the steps discussed in the Producer Thread section of this article.

```
 1  int main(int argc, char** argv)
 2  {
 3      // Process command line arguments
 4      if ( argc < 3 || argc > 4 ) {
 5          printf("usage: %s <workers> <port> <ip>\n", argv[0]);
 6          exit(-1);
 7      }
 8      int workers = atoi(argv[1]);
 9      int port = atoi(argv[2]);
10      string ip;
11      if (argc == 4) {
12          ip = argv[3];
13      }
14
15      // Create the queue and consumer (worker) threads
16      wqueue<WorkItem*>  queue;
17      for (int i = 0; i < workers; i++) {
18          ConnectionHandler* handler = new ConnectionHandler(queue);
19          if (!handler) {
20              printf("Could not create ConnectionHandler %d\n", i);
21              exit(1);
22          }
23          handler->start();
24      }
25
26      // Create an acceptor then start listening for connections
27      WorkItem* item;
28      TCPAcceptor* connectionAcceptor;
29      if (ip.length() > 0) {
30          connectionAcceptor = new TCPAcceptor(port, (char*)ip.c_str()
31      }
32      else {
33          connectionAcceptor = new TCPAcceptor(port);
34      }
35      if (!connectionAcceptor || connectionAcceptor->start() != 0) {
36          printf("Could not create an connection acceptor\n");
37          exit(1);
38      }
39
40      // Add a work item to the queue for each connection
41      while (1) {
42          TCPStream* connection = connectionAcceptor->accept();
43          if (!connection) {
44              printf("Could not accept a connection\n");
45              continue;
46          }
47          item = new WorkItem(connection);
48          if (!item) {
49              printf("Could not create work item a connection\n");
50              continue;
51          }
52          queue.add(item);
53      }
54
55      // Should never get here
56      exit(0);
57  }
```

[Line 4-13] The number of consumer threads, the listening port and the server IP address are specified on the command line. Note that the specification of a listening IP address is optional.

[Line 16-24] Create the work queue object and the number of `ConnectionHandler` threads specified on the command line. For each handler call the `Thread::start()` method ultimately calls the `ConnectionHandler::run()` method.

[Line 27-38] Create the `TCPAcceptor` object for the listening port and IP address, if specified, or just the listening port if the IP address is not specified. Note that specifying the server IP address will cause the `TCPAcceptor` to listen for connections on the network interface to which the IP address is bound. When no IP address is specified, the `TCPAcceptor` listens on all network interfaces.

[Line 41-53] Called in an infinite loop, `TCPAcceptor::accept()` blocks until it receives a connection. For each connection a `WorkItem` is created and passed a pointer to the resulting `TCPStream` object then placed onto the work queue.

Client Application

The client application code resides in a single file *client.cpp*. It starts with the header files we need from C/C++ environment and the interfaces for the `TCPConnector` class. The client simply makes a connection, sends a message to the server and waits for the server to echo it back. This action is performed twice. In both cases the message sent and received back is displayed to stdout.

```
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "tcpconnector.h"

using namespace std;

int main(int argc, char** argv)
{
    if (argc != 3) {
        printf("usage: %s <port> <ip>\n", argv[0]);
        exit(1);
    }

    int len;
    string message;
    char line[256];
    TCPConnector* connector = new TCPConnector();
    TCPStream* stream = connector->connect(argv[2], atoi(argv[1]));
    if (stream) {
        message = "Is there life on Mars?";
        stream->send(message.c_str(), message.size());
        printf("sent - %s\n", message.c_str());
        len = stream->receive(line, sizeof(line));
        line[len] = NULL;
        printf("received - %s\n", line);
        delete stream;
    }

    stream = connector->connect(argv[2], atoi(argv[1]));
    if (stream) {
        message = "Why is there air?";
        stream->send(message.c_str(), message.size());
        printf("sent - %s\n", message.c_str());
        len = stream->receive(line, sizeof(line));
        line[len] = NULL;
        printf("received - %s\n", line);
        delete stream;
    }
    exit(0);
}
```

Test Server and Client

Build

Get the code for the project from Github – https://github.com/vichargrave/mtserver. You'll also need the code from these repositories:

- threads - https://github.com/vichargrave/threads
- wqueue - https://github.com/vichargrave/wqueue
- tcpsockets - https://github.com/vichargrave/tcpsockets

Place all the directories in the same folder then cd into *mtserver/* and run *make*. This will build the client, server and all dependencies across the folders.

Run

First run the server listening on TCP port 9999 and with 5 consumer threads like this:

```
$ ./server 5 9999 localhost
thread 4426719232, loop 0 - waiting for item...
thread 4430274560, loop 0 - waiting for item...
thread 4429737984, loop 0 - waiting for item...
thread 4428664832, loop 0 - waiting for item...
thread 4429201408, loop 0 - waiting for item...
```

Next run a series of clients like this:

```
$ client 9999 localhost; client 9999 localhost; client 9999 localhost
```

Six messages, two by each client, are sent to the server. Both the original and echoed messages are printed to stdout. The output of the series of client apps should look like this:

```
sent - Is there life on Mars?
received - Is there life on Mars?
sent - Why is there air?
received - Why is there air?
sent - Is there life on Mars?
received - Is there life on Mars?
sent - Why is there air?
received - Why is there air?
sent - Is there life on Mars?
received - Is there life on Mars?
sent - Why is there air?
received - Why is there air?
```

The server output should show the thread status and the messages it receives from the clients. Note that different threads handle different connections indicating the server is distributing the work items as expected.

```
thread 4426719232, loop 0 - got one item
thread 4426719232, echoed 'Is there life on Mars?' back to the clien
thread 4430274560, loop 0 - got one item
thread 4430274560, echoed 'Why is there air?' back to the client
thread 4429737984, loop 0 - got one item
thread 4429737984, echoed 'Is there life on Mars?' back to the clien
thread 4428664832, loop 0 - got one item
thread 4428664832, echoed 'Why is there air?' back to the client
thread 4429201408, loop 0 - got one item
thread 4429201408, echoed 'Is there life on Mars?' back to the clien
thread 4430274560, loop 1 - waiting for item...
thread 4426719232, loop 1 - waiting for item...
thread 4430274560, loop 1 - got one item
thread 4430274560, echoed 'Why is there air?' back to the client
thread 4429737984, loop 1 - waiting for item...
thread 4428664832, loop 1 - waiting for item...
thread 4429201408, loop 1 - waiting for item...
thread 4430274560, loop 2 - waiting for item...
```

Author: Vic Hargrave

Multithreaded Work Queue Based Server in C++ was posted on **March 14, 2013** at **5:15 pm** in Programming and tagged as C/C++, Design Patterns, Network, Pthreads, Sample Code, Server, TCP/IP. It was last modified on **August 25, 2013** at **10:46 am**. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response or trackback from your site.

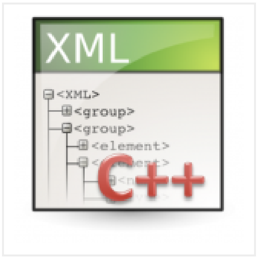
Share on Twitter, Facebook, Delicious, Digg, Reddit

«Previous Post Next Post »

Related Posts

Condition Variable Class in C++

Mutex Class in C++



XML Parsing with DOM in C++

10 Comments

Divyang Patel on March 26, 2013 at 2:51 am.

Thanks Vic. That's nicely done oo multithreading server. I've one question though, what are your thoughts about creating thread pool beforehand to serve?

-Divyang

Reply

Vic on March 26, 2013 at 8:38 am.

Well the consumer threads are created in advance of any connections so they are a thread pool. You can specify the number of consumer threads to add to the pool on the server command line. Thanks for your comments.

Reply

David on April 28, 2013 at 7:03 am.

Excellent article , very helpfull.

Reply

Mark on May 22, 2013 at 8:35 pm.

Need netinet/in.h , netinet/ip.h for compile in Linux/FreeBSD

Reply

Vic on May 22, 2013 at 9:12 pm.

The project includes netinet.h/in.h in tcpconnector.h and tcpacceptor.h. netinet/ip.h is not required.

Reply

Pradeep on October 2, 2013 at 11:51 pm.

Great explanation.

Could you please throw some light on handling the corruption in consumer threads?.
Also let us suppose the consumer thread is crashing every-time it try to process particular kind of job. ?

Reply

Vic on October 6, 2013 at 8:36 am.

It depends on the particular problem you are talking about. By crashing I assume you mean the thread core dumps. If that happens it takes the whole process down and all the other threads the process contains. Again it is difficult to address this question generally. There are lots of things that cause processes and their threads to crash.

Reply

Pradeep on October 2, 2013 at 11:52 pm.

correction above…particular kind of item in the queue ?

Reply

Vic on October 6, 2013 at 8:33 am.

Any type of item can be stored in the queue because it is a template collection. However, you should store pointers to object in the queue so the item is not destroyed when each item is removed from the queue.

Reply

Bir on October 16, 2013 at 8:30 am.

Vic, all your articles are excellent and very informative. Thanks for explaining multi-threading so lucidly.

Reply

Leave Your Comment

Your email will not be published or shared. Required fields are marked *

Name *

Email *

Website

Please verify you are a real person.

Type the text

Privacy & Terms

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>`

Submit