Web Server Core Design (Part 1)

BY EVAN OLDS

CPT S 422

Needed Elements

- Proper concurrency model
- Ability to have a reusable request parser that can support different types of handlers
  - We don’t want to have to re-write the core web server stuff when we make new web “applications”
  - Most web applications are not actually standalone apps. Rather, they are almost like plugins that get request objects from a core server
    - IIS
    - Apache

Motivation for Concurrency

- Web server needs to handle multiple requests at once
- Can’t wait to complete download of 1 GB file before letting any other clients connect
- Server WILL only listen on 1 port, so in a sense it does accept one client at a time
  - More of a literal sense here: at any given moment we’re blocked on a single AcceptTcpClient call in the “listening thread”
- What logical question does this lead you to ask about AcceptTcpClient?

What AcceptTcpClient Gives Us

- "After accepting a TcpClient with AcceptTcpClient can I send the client object off to some other thread and go right back to another AcceptTcpClient call on the listening thread?"
- Yes. AcceptTcpClient, it it succeeds, gives us a socket that we can communicate with while accepting another client on the same port as the previous AcceptTcpClient call.
  - The TcpClient returned basically communicates on a different port after it has been accepted, so the original remains open

Process requests on new…?

- So we can read from and write to a TcpClient while waiting to accept another
- This means we can process the request on a new thread (if one is available)
    - Threads from thread pool should be waiting, ready to handle requests from accepted TCP clients
- We WILL use threads and a thread pool for our server, but this isn't necessarily what's used on the average industry web server
- What the other option for a concurrency mechanism that could process a request?

Threads vs. Processes

- Many web servers will spawn a new process to deal with a request
  - Why?
  - What does using a process over a thread allow us to do?
  - So that we see both pros and cons, what would be the advantage of threads only instead of processes?
  - Discuss in class

Threads vs. Processes

- Reasons threads are better
  - Faster. Thread pool gives us quick access to threads.
  - Easier to manage a thread pool vs. a process pool. Don't need IPC. Also, the process-based-handling servers actually are not likely to use a process pool because of reasons that should have been discussed from the previous slide (need to spawn process with an account that's related to the permissions associated with the request)
- Reason processes are better
  - Process permissions allow for use of operating system security. Restricts the request from accessing areas of the file system that it should have access to (i.e. "sandboxed")
  - If process crashes, not as big of a deal as a thread crashing.

Listening Thread Logic

```
while (server_active) {
  1. Accept TCP client
  2. Send client to a thread pool thread to be processed
}
```

- More to come in this logic when we start discussing how to terminate the server
  - Raises the question of how we break out of blocking accept call

Worker Thread Logic

```
while (server_active) {
  TcpClient client = sharedBlockingCollection.Take();
  // Read request
  // Write response (if applicable)
  // Close client
}
```

- Is it that simple?
- Questions we haven't addressed yet:
  - What do we write in the response?
  - How do we design a reusable server code that allows for different types of handlers to be written?

Worker Thread Logic

- Consider:

```
while (server_active) {
  TcpClient client = sharedBlockingCollection.Take();
  MyWebRequest req = PackageRequest(client);
  MyWebHandler handler = FindHandlerFor(req);
  handler.Handle(req);
  // Close client
}
```

Parsing and Packaging the Request

- Want the ability to have different types of handlers
- Don’t want each handler to have to do the following:
  - Parse HTTP request to find out whether or not it’s even valid
  - Parse HTTP request pieces to get the method, URI and version as separate strings
  - Parse HTTP headers and put them in some easily accessible collection
  - Put the body of the request into a simple Stream object
- Core logic, in a “PackageRequest” or “BuildRequest” function does all the above.
  - Send “simple” (as it can be) request object to handler
  - Also implies that the core logic manages a collection of handlers and determines which ones handle which requests

Request Object

- Request object that we pass to handlers contains:
  - Parsed info from the first line of the request
    - Method (GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, or PATCH)
    - Request target (URI)
    - HTTP version
  - Headers parsed and packaged (what data structure is good to store a collection of header names/values?)
  - Stream for the body
- Request object may also provide utility functions to write responses

Request Object

```
public class Request
{
  Stream Body;

  ConcurrentDictionary<string, string> Headers;

  string Method;

  string RequestTarget;

  string HTTPVersion;

  public long GetContentLengthOrDefault(long defaultValue)
  {
    // ??
  }

  public Tuple<long,long> GetRangeHeader()
  {
    // ??
  }

  System.Net.Sockets.NetworkStream Response;

}
```

Request Object Body Stream

- There is content in an HTTP request before the body (request line and headers)
- Want a single Stream object for the body
  - This stream doesn't contain any data before the body
  - First byte in stream is first byte of body content
  - Wouldn't be convenient if it were any other way
- If we're implementing a BuildRequest function that needs to set the body stream, where do we get it from?
  - Can't just set it to the network stream. Why not?

Request Object Body Stream

- If the position in the network stream were right at the beginning of the body, then we COULD just set it as the body stream
  - Shows us that the body stream is likely not going to support seeking
  - Forward-only reading
- What if we read into a buffer and got the entire first line, all the headers, and some of the body?
- How do we construct a Stream that includes the part of the data we've already read as well as content from the NetworkStream that has yet to be read?