# Simple Web Server Project

##### **Marina Kraeva, Brian Collins**

## Tactics Implemented

### Security tactics

* Monitor incoming requests so that we can catch DOS attacks
  + If there are more than a certain number of requests by a client, the server is degraded to sending 503 errors: Service Temporarily Unavailable. If even more requests come from the same client, the server starts dropping the request from it.
* Limit access
  + Access is limited: we define at what port the server is started, so a client must know this port in order to connect to us
* See Availability tactics below

### Performance tactics:

* Limit event response
  + Event queue: Incoming requests are not dropped if they come before the previous one has been processed, they are kept in a queue and handled on a First-In-First-Out principle, separately for each connection
  + Maintain persistent connection: Connection is not closed immediately after handling a request. Instead, there is a 15 second period of wait. If no new requests come during that time, or the client closes the connection, then the server closes it.
* Maintain cached data in memory (clear up every midnight)
  + Data of highly requested objects is stored in a singleton “FileTracker” class for easy access. This avoids the overhead caused by constantly fetching that data from the file system.
* Dynamic Priority Scheduling (based on how many requests the user has sent us so far, for how long the request has been waiting etc.)
  + We use a thread pool which switches the threads responsible for managing the requests. Each thread is responsible for reading the HttpRequest objects and forming appropriate responses.

### Availability tactics:

* Monitor system activity – a Server Monitor counts
* Exception handling
  + We catch exceptions caused by connection failures and such and let our system handle them so that it does not crash
* Degradation (send 503 Service Unavailable response)
  + We will send 503 type responses to a user when that connection or the whole system is over acceptable request limits.

## Availability Comments

### Original Web Server

Thread pool: 100

Connection interval: 50 ms

File size: 26KB

# of attackers: 2

Times to failure:

52.315s

78.589s

22.799s

ATTF: 51.234s

ATTR: 1.5s (to react and click to restart the server)

a = (ATTF - ATTR)/ATTF \* 100%

a = 97.072%

### Modified Web Server

We ran the server under the same conditions as the original and no crash or error was encountered, although the attack was left running for much longer time.

We also ran the attack with requesting a larger file (156KB) and then a much larger file (1.5MB) with no crash or error.

For the same conditions used in the testing of the original server, our modified version has availability 100%.

## Performance Comments

##### Original Web Server

Thread Pool: 100

Connection Interval: 500ms

Fetched file size: 7KB (ConnectionHandler.java)

Connections/Second: 12

The measured throughput during a DoS attack is 12 requests per second.

We have implemented persistent connections. That means that for 50 requests on the same connection we save 49\*RTT that would be used for establishing connection in the original version of the server.

We have also implemented the optional 304 response: Not Modified. For a 5MB picture that is requested 10 times using a conditional GET request, we would gain the time to send 45MB of data back to the client, since we would only send 304 response and not 200 OK response containing all the data.

The FileTracker keeps in memory frequently requested files, which reduces time for response since there is less overhead from I/O operations.

## Security Comments

The original version of the server does not implement security tactics, i.e. the server crashes relatively quickly under a Denial of Service attack. Its MTTF (mean time to fail) is 51s, after which it crashes and needs to be restarted.

Our implementation keeps track of when there are too many incoming connections. If we cannot handle them, we degrade the server to send 503 Service Unavailable responses and then start dropping requests, but only the ones from the client that has been determined to have unreasonably many requests. In effect, the server should be able to recover by itself if given some time. A restart would fix the issues, too.