CircuitBreaker and Microinvader

By Glauco Reis (gsreis@br.ibm.com.br) WW Competitive Migration Team

When the subject is Microservices, usually there are more than fragmentation of services and isolation using containers in discussion. A set of new paradigms (some disruptive) and programming patterns are on table, and we should take care of it. If not respected, we can have old problems of distributed computing and services amplified.
One of these rules are related to Microservices' availability.
Distributed computing premises that several components are going to run splitted along the servers, with communication between them. It is impossible to grand 100% of execution for all services, all the time. This is a extension of distributed computing fallacies
(https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing).
When using MicroServices, it is desired some sort of adaptedness, and even in case of some failure, others should adapt itselves and execute gracefully.
When a Microservice is not running, because maintenance issues or by some network latency, other Microservices trying access will fail also. This is normal. The question is that until failure happen, there are some latency until client be aware of that. Microservices uses frequently REST and HTTP, usually synchronous requests, and after timeout's trigger everything goes on (perhaps with errors). This waiting time can compromise entire system, and the user can have a bad perception of the system.
In the past, we had a concept of "all or nothing", and errors should be evident for entire system, because some action must be easily identified. The fast action is still needed today (of course), but there is a general perception that is better to run as best as we can, even when error happens. The user experience has a new level of importance on newer systems. Errors should be handled in second plane, not using main flow of execution.
How to reduce timeout latencies between Microservices calls?
There is a pattern that could help, presented by the first time on the book "Release It!" from Michael Niggard. The name is circuit breaker, like the ones controlling energy on houses. Basically, when electric current exceeds some threshold, circuit breaker switches to disabled. In computing is not possible to switch based on levels of current, but we can measure number of failures. If some number of sequential failures happen, CB is disabled. In the real world, the enabling process is manual, but in computing systems we should not count on that. So after some delay the CB turn on again. It is expected after this time the latency or problem has gone.
A CB implementation could use an Adapter (GOF95), enveloping requests on a class that will handle errors.
Original calls for Microinvader are:

```
private String  callRest(String urlin) {
        try {
            Client client = ClientBuilder.newClient();
            return client.target(urlin).request(MediaType.APPLICATION_JSON).get(String.class);
        }catch(Exception e) {}
        return "[]";
}
```

The evolving class can have some parameters for lifecycle control (number of failures to disable and timeout to arm again).

```
// removed some code to save space
public String callRest() {

               try {
                  if (!breaked) {
                      return client.target(urlin).
                       request(MediaType.APPLICATION_JSON).get(String.class);
                   }
                   else {
                      if (System.currentTimeMillis() - started_timeout >= timeout) {
                             breaked = false;
                             tries_count = 0;
                      }
                  }
                  }catch(Exception e) {
                      tries_count++;
                      if (tries_count >= tries) {
                         tries_count = 0;
                         breaked = true;
                         started_timeout = System.currentTimeMillis();
                  }
                }
                return "[]";
        }

}
```

The principle is that after disarm CB, system will be faster for a while, until the system stabilize. Perhaps the better implementation for this pattern should be an annotation, and a lot of discussion groups are working on that.
For my implementation, I've created a "forest" of CBs, each one taking care of one URL.

```
public class CollectionCircuitBreakers {

       private Hashtable<String, CircuitBreaker> vector = new Hashtable<String,
CircuitBreaker>();
       private int tries;
       private long timeout;
       public CollectionCircuitBreakers(int tries, long timeout) {
              this.tries = tries;
              this.timeout = timeout;
       }

       public String callRest(String urlrest) {
              CircuitBreaker cb = vector.get(urlrest);
              if (cb == null)
              {
                      cb = new CircuitBreaker(tries, timeout, urlrest);
                     vector.put(urlrest, cb);
              }
              return cb.callRest();
```

```
        }
}
```

The video show execution of game without CB by left side, and with this CB implementation by right side. It is possible to see that right side is faster (get the end first) and some smooth caused by latency (spaces between enemies) are not perceptible as well.

To execute this code, just download microinvader_cb.zip. There are two directories inside it (microinvader and microinvadercb). Just run “mvn install” on each one, and two servers will be created (ports 9081 and 9082). Start the servers as usual (**/target/wlp/bin/server start Microinvader**) and open two browsers (http://localhost:9081/space-1.0 and http://localhost:9082/space-1.0).
As usual all source code inside the zip file, for further exploration.