Service Discovery and Clustering for .NET developers
Who are you?

• Software Developer for over 20 years
  • Worked mainly for ISVs
    • Reuters, SunGard, Misys, Huddle
  • Worked for a couple of MIS departments
    • DTI, Beazley

• Microsoft MVP for C#
  • Interested in OO, SOA, EDA, Messaging, REST
  • Interested in Agile methodologies and practices

• No smart guys
  • Just the guys in this room
Intelligent collaboration for the Enterprise
Agenda

• Point-to-Point
  • The Fallacies of Distributed Computing
  • Timeout, Retry, and Circuit Breaker
  • Load Balancing

• Discovery
  • Client-Side Discovery
    • Static: Local Registry
    • Dynamic: Registrar
    • Dynamic: Self-Registration
  • Health Checks
    • Registering a health check for a service
  • Server-Side Discovery
    • Load Balancers
    • Load Balancers and Service Registration

• Clustering: An Introduction

• Q&A
Point-to-Point
EAI with HTTP

REST

Easy to deliver as requires no MoM within the organization.

Communication is point-to-point.

Communication is frequently synchronous, but highly cacheable.

Can use status 202 and monitor a link for asynchronous approaches.

Can be high-latency, especially if service A, calls B, calls C etc. Minimum time is controlled by latency of hops.

Can require considerable effort to build a client of the API that supports desired QoS attributes.

No built in support for competing consumer or at least once patterns of message delivery.
Demo: Point-to-Point

In this demo we will review point-to-point integration between an HTTP Server and a Client.

Although this model is simple, we will demonstrate its obvious weaknesses.

Next we will look to understand these issues, and solutions to them.
The Fallacies of Distributed Computing

The network is reliable.
Latency is zero.
Bandwidth is infinite.
The network is secure.
Topology doesn't change.
There is one administrator.
Transport cost is zero.
The network is homogeneous.
Fault Recovery
The Timeout Pattern

Provide a timeout on any inter-process operation.

The timeout pattern is usable with any client-server request.

The timeout pattern is also usable with thread resource pools.

Any operation that blocks a thread, should have a timeout

Usually a timeout is a parameter on an API call that indicates how long to wait [in milliseconds] for the operation to complete before returning control to the caller

If you are using a third-party API that has no timeout, you may need to make the request on a thread that does have a timeout, and when that thread times out assume that the operation failed.

• Timeouts are a scaling issue. Because an API call without a timeout consumes resources, a stalled network call, or thread, prevents other requests from being executed. You might not notice the failure to use timeouts on a system with abundant resources, but as your resources dwindle failure to use timeouts to free resources will cause pain.

• Assume that a timeout will occur at some point

• Asynchronous operations still need to time out. All the caller may not be blocked, resources are still being consumed to manage the asynchronous operation that will not be released until the operation aborts.
The Retry Pattern

In the presence of unreliable calls between two components our first approach to achieving a high quality of service is to recognize that many such failures are transient: a timeout because a resource is busy, temporary loss of connectivity, the loss of one node that will be replaced by failover to another.

In this case the fault is self-correcting, the node comes up, the load on the database or server declines and their is capacity for our call, or network connectivity is restored. This means that our call will succeed if we retry after a delay to allow the transient fault to resolve.

The Retry pattern is simply that if the call fails, we can try again. It is important to have an upper bound on retries in case a fault that appears transient is not. See Circuit Breaker as well.

• If the fault is permanent, and unlikely to succeed, for example a login failure due to invalid credentials – don’t retry.

• If the fault is caused by a rare event, for example packet loss or other corruption, consider an immediate retry as the server may be able to respond.

• If the fault is caused by load, such as SQL Timeouts, or 429 Too Many Requests, then back-off for a period before retrying. Failure to observe this can lead to an Application Denial of Service Attack i.e. we overload a struggling server.

• In the case of a 429 for example, the Retry-After header will tell you how long to back off for.
The Circuit Breaker Pattern

The Circuit Breaker pattern prevents an application from executing an operation that is likely to fail, thus freeing up resources that would otherwise be consumed waiting for a timeout and retry cycle to occur.

Once normal service has been restored to the server or resource pool the circuit breaker pattern allows detection of the resumption of service.

A Circuit Breaker acts as a proxy to operations that can fail. It has one of three states:

Closed: Requests are routed as normal, on a failure a counter is incremented and if the threshold is exceeded within a time limit, the circuit breaker opens.

Open: No calls are allowed, and are failed automatically by the proxy. After a specified time interval the circuit breaker is moved to half-open.

Half-Open: A call is allowed. On a failure the breaker moves to Open, on a success it moves to Closed.

Introduce Redundancy

Load Balancing

Load Balancing distributes work across multiple resources. This provides redundancy in the event that one instance is not available, allowing clients to meet their request from another service.

We have two groups of options:

- **Hardware of Software Load Balancers: their goal is to listen on a port and forward request on that port to a set of services – the pool of servers appears as one service to the client.**
  - A load balancer can itself become an availability concern, unless it is deployed in a pair.

- **Round Robin: the client is made aware of the pool of servers and is chooses from amongst them.**

Persistence

- If you introduce redundancy you have to be aware of state across those servers.
  - The easiest model is to be session-aware i.e. state is in a database and loaded with each request.
    - The database becomes the bottleneck.
    - You may need a “leader and follower” approach to provide redundancy.
  - You can also choose to have a sticky-session which forwards the request to the same server until it is completed.
    - An instance of the server stores the client’s session data.
    - This may not just be in memory, but local storage on that server.
    - The danger is that we lose a given server instance and the associated data, effectively cancelling a user’s session.

- A Web Farm is a classic example of using redundancy to improve availability – in the event of load or failure
Discovery
Where is my server?

Discovery

Redundancy introduces the question: how do I know who to talk to?

Instead of a single server, we now want to talk to a pool of servers, so that we can find an alternative if one of our servers is unavailable.

Discovery is the process of a client retrieving all the servers that make up a pool, which allows it load balance between them.

Client or Server-Side Discovery

- **Client**
  - We can choose to make information about the pool available to the client directly.
  - The client then implements any algorithm to choose between servers from the pool.
  - We may choose to give different client instances different pools, so as to spread load amongst our servers.

- **Server**
  - We can choose to put the pool behind server-side load balancing software or hardware, and have the client route via the load balancer.
  - To keep the pool fresh we may want to provide a health check, which confirms whether or not a service is alive, and route requests to it as appropriate.
  - In both cases, the major issue is keeping the client’s pool up-to-date. With static approaches, an operator needs to intervene to create a new pool, whereas dynamic approaches adjust.
Client-Side Discovery
Local Registry

Simplest option.

The simplest option, in that it requires no additional tooling, is for the service to read a local configuration files to obtain a pool of servers.

The client reads that file to acquire it’s own pool of servers

The client then needs to determine which server to pick, usually via a Round Robin Algorithm.
Demo: Local Registration

In this demo we look at using the configuration file of the application to provide a pool of servers which we can talk to, if our first server fails.

We will also look at how this approach copes with the loss of a server.
Advantages

• Easy to implement
  • Most developers already have the skills to work with a configuration system(s) for their platform.
  • The pool of resources is a simple list, and the process of iterating over the list is straightforward.
  • You can load balance using a simple Round-Robin algorithm.

Disadvantages

• In environments with auto-scaling or manual scaling frequent changes to service locations are problematic as they require re-distributing the application to distribute the configuration.
  • Configuration has to be updated for new service deployments, as well as for hosts failing or being added/replaced.
  • No health checks, so we try services that are failing, and re-route once they fail.
Server Registration

We can store configuration on a dedicated server.

The simplest form of registration server is just a key-value store.

- You can use Redis as a registration server if it is already part of your infrastructure.
- Of course, your registration server becomes a new point of failure, if clients cannot get hold of it, they can find the pool of servers.
- So your registration server must itself use redundancy to avoid availability concerns.
- The usual redundancy model chosen is a cluster.

Registration Servers

- **Zookeeper**: CP, register on start-up, deregister when done, list currently registered services, subscribe to notifications on ones that leave. Java and C libraries only.
- **AirBnB SmartStack**: CP, Hadoop + Zookeeper
- **Netflix Eureka**: AP, registration server with cached client that polls, register service on start-up and send heartbeat. REST and Java, designed for use on AWS
- **etcd**: CP, key-value store, Raft for consensus, has an HTTP+JSON API
- **SKyDNS**: CP, Raft for consensus, has both HTTP + JSON API and DNS support.
Service Registrar (Sidecar)
Demo: Service Registrar

Service Registrar Registration.

In Service Registrar Registration another process (sometimes called a sidecar) does the heavy lifting around registering the services.

Although we only run this once in the demo, in principle the registrar can run as a scheduled job and thus continually update registrations, or listen for events that tell it new services are available to register.

In practice this relies on the service reloading the configuration automatically when an operator updates it for new registrations.

We could run a registrar for many services as here, or run one registrar per service, and schedule it to update the service at intervals.
Advantages

• Easy to implement
  • The pool of resources is a simple list, and the process of iterating over the list is straightforward.
  • You can load balance using a simple Round-Robin algorithm.
  • Services are not dependent on the registration server implementation that we are using. So only the client and registrar need to be change if we decide to switch that out.
  • This also means we can use this approach for services that we did not write ourselves.
  • You can use projects like Registrator if you are working with Docker, which inspects new Docker containers for servers and registers them (supports multiple OSS Service Registries)

Disadvantages

• In environments with auto-scaling or manual scaling frequent changes to service locations are still problematic as they require re-configuring the registrar to update the registration server with the new locations.
• We now have a dependency on our Service Registrar and Registration Service to run in production. So we have to worry about make these highly available as well, or they become our point of failure.
Self-Registration
Demo: Self-Registration

Self-Registration

In self-registration a server simply registers itself with the registration service at startup and de-registers itself at shutdown.

This means that there is no sidekick service required, services simply take themselves in and out of the pool as the come online and go offline.

The client queries the registry for the pool as before.
Advantages

• Handles dynamic environments
  • As a server is self-registering so it adds itself to the pool.
  • You can load balance using a simple Round-Robin algorithm.

Disadvantages

• Service now has a dependency on the registration service. If we wish to change our registration service, we need to update services not sidekicks.
• We have more dependencies on the Registration Service, so again, we need to ensure that it is clustered.
• Does not work for apps that we do not own and thus cannot make self-register
• No health checks, so we try services that are failing, and re-route only once they fail.
Health Checks
Health Check
Health Checks

Is my server alive?

A health check reduces the problem that we may round-robin to a dead server (as we do in our examples so far).

- If the service is not valid, we can take it out of the pool when we look for a server to use.
- A health check can be:
  - A custom script that the registration server calls
  - A GET against an HTTP endpoint, 2xx is alive, anything else is dead

A call by the server to the registration service to renew a lease before it times out.

We don’t show here that you would want to poll on a regular basis and update your list.
Server-Side Discovery
Hardware or Software Load Balancer

**Load Balancers**

A Load Balancer (LB) is a common solution to making a pool of servers available to clients.

A service is registered with an IP address (sometimes called a Virtual IP or VIP) that is given to clients; that IP address represents not (necessarily) an individual server but a pool of servers.

The LB then uses a scheduling algorithm such as Round-Robin to distribute requests to the pool.

An LB can support ‘sticky’ sessions where the server holds state.

The LB can use health checks to determine if servers should be in the pool.
Advantages

• It’s not a software developer problem to do load balancing, we just buy some dedicated kit.
• An LB typically has some useful additional features such as SSL termination, priority queueing, DDOS protection, firewall etc.

Disadvantages

• The LB itself could become a single point of failure, so we need to have active-active or active-passive redundancy.
• The LB still needs to be configured with the IP addresses of the server in the pool. This may be problematic as we have to understand how servers will register with the LB when they spin up and shut down.
• An LB is mostly used in an external facing role. To use the LB internally it either needs to be dedicated to that role, or expose the services externally. For example, to use an AWS ELB in an internal load-balancing role you need to use a Virtual Private Cloud (VPC), not EC2 classic.
Service Registration and Load Balancers

Proxy Load Balancers

It is possible to have the Load Balancer and Service Registration work together.

In this approach the Service Registry updates the Load Balancer when a new service appears, or one fails its health checks and should be removed.

For example Consul-Template can be used to update configuration files from a template using the values in the Service Registry (service of just key-value pair). This can then update the configuration of a Varnish, nginx, or HA-Proxy proxy acting as a load balancer.

https://github.com/hashicorp/consul-template
Clustering: An introduction
Cluster Attributes

A cluster makes many nodes appear to be one. It is used to provide availability.

What kinds of cluster are there?

- **Load Balancing**: In essence, what we have discussed so far. Used to distribute workloads, using a scheduling algorithm.

- **High Availability**: Redundant nodes, which provide service when components fail.

Typically the main issue is whether we care about propagating state from one node to another.
CAP Theorem


It said that, in a distributed system, of these three properties, you can pick any two.

Consistency (all nodes agree on the state)

Availability (you get a response from one of the nodes when you make a request)

Partition tolerance (if nodes cannot communicate, or communicate at too high a latency, the system continues to offer service)

Usually short-handed to you can choose CP, or AP (or CA if P indicates catastrophic failure)

You should also see the 2012 paper: CAP Twelve Year later

Distributed Systems for Fun and Profit
How does this affect us?

The Chaos Monkey Test

The chaos monkey test, devised by Netflix as part of its Simian Army tools, deliberately kills nodes in production.

The goal is to force teams to build software that survives the loss of nodes.

It enforces understanding the fallacies of distributed computing – you must account for the loss of a node.

Service Discovery is a tool to allow you to have multiple nodes running your services, so that you can survive the loss of nodes and recover.

It is necessary but not sufficient, as to be HA you need to account for your data as well.

High Availability

• Load Balance your services
  • This is the strategy that we have discussed so far. Load Balancing the services enables you to scale to meet load, and remain available
  • High-Availability Load Balancer or Service Registration
    • In order to load balance we need to hold state – the routing of clients to servers – and we need to replicate that state between nodes so that if we lose a node we continue to operate.
    • Most Service Registration tools are CP. They rely on client (or LB) locally caching and thus permitting stale definitions, for availability, over allowing inconsistent data on the node. (Some let you ask for eventually consistent data and thus become AP). Netflix Eureka is AP, it favours being able to talk to a Registration service over it being available.

• Make your data HA
  • Your data needs to live in an HA cluster
  • In essence you need leader-follower, with the follower reading logs from the leader and applying locally, with a consensus protocol used to ensure agreement on state and a process to elect a new leader.
Q&A