WHITE PAPER

Using Hazelcast with Microservices

By Nick Pratt

Vertex Integration

June 2016

WHITE PAPER

Using Hazelcast with Microservices

TABLE OF CONTENTS

TABLE OF FIGURES

ABOUT THE AUTHOR

Nick Pratt is Managing Partner at Vertex Integration LLC. Vertex Integration develops and maintains software solutions for data flow, data management, or automation challenges, either for a single user or an entire industry. The business world today demands that every business run at maximum efficiency. That means reducing errors, increasing response time, and improving the integrity of the underlying data. We can create a product that does all those things and that is specifically tailored to your needs. If your business needs a better way to collect, analyze, report, or share data to maximize your profitability, we can help. Mr. Pratt may be reached at npratt@vertexintegration.com and telephone +1 (914) 935-7890.

1. Introduction

1.1 WHAT IS A MICROSERVICE

A Microservice (or 'service') is *any piece of a system regardless of size that has a consistent interface and is independent*. Each piece can stand on its own at deployment and runtime.

James Lewis & Martin Fowler elaborate on this definition to include the style of architecture that include Microservices: *the Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API*[1].

2. Our experience using Hazelcast with Microservices

For Vertex's in-house Commercial Mortgage Lending Platform, we wanted a platform that was easy to build, maintain, extend, operate and diagnose. We started out knowing that we would run what we built – there was no throwing code over a wall. While we initially started out with a monolithic application, we transitioned to a service based architecture over time (after about 18 months) as we increased the functionality beyond the core viable product and started to implement customer specific features.

There were three main reasons for this shift to Microservices architecture:

1. Restoration time after failure
2. Ease of adding new functionality without impacting users
3. Ease of use on a developer machine

We elected to use Hazelcast as the backbone of our Microservices architecture. Hazelcast is simple, a single JAR with no external dependencies. One of the core values of Hazelcast from its inception was simplicity and ease of use, and that still exists today. When you move an application or system into a distributed environment, having a simple and configurable backbone makes the task a lot easier.

Hazelcast is a a library and over the years at Vertex, we've come to highly value libraries far more than frameworks. In our experience, libraries are just more simple to use and understand and make the development process more efficient. That said, we also live by the rule that if we can't get any software (library or framework) running within 15 minutes, we move on. Hazelcast also met this standard – in our initial development spike, Hazelcast passed the 15 minute "test" in under 2 minutes. This simplicity and ease of use is a hallmark of Hazelcast and the reason it adapts well to Microservices architecture.

We have also leveraged Hazelcast ('HZ') because of its highly scalable In-Memory Data Grid (e.g. IMap and JCache). Hazelcast also supports other standard JDK Collections such as Lists, Sets, Queues and some other constructs such as Topics and Ringbuffers that can be easily leveraged for inter-process messaging. All of these attributes can offer functionality that is highly desirable within a Microservices platform.

2.1 DEPLOYMENT

Hazelcast can be deployed two ways:

- Embedded, where you add the JAR to your application and the Hazelcast Member instance is embedded in your application
- Client-Server, where Hazelcast is run on its own. Here you can run Hazelcast from the command line, a Docker container or a cloud deployment such as AWS and Azure.

[1] https://www.thoughtworks.com/insights/blog/microservices-nutshell

The client-server method of deployment is commonly used in more traditional environments and is not recommended for a Microservices architecture.

2.1.1 EMBEDDED

For a Microservices platform, Hazelcast is best deployed standalone as a cluster either on its own or with a thin admin/control-plane on top (which is what we did with our platform).

Each Microservice then uses a Hazelcast client which can be started and stopped in isolation without impacting the Hazelcast cluster, either from a storage or messaging standpoint. If Hazelcast server nodes ("Members") were being constantly added and removed, as would be the case if each Microservice instance embedded a Hazelcast server node, certain challenges would arise. Data within the Hazelcast cluster Members would be constantly rebalanced or potentially lost if multiple Services were taken down concurrently.

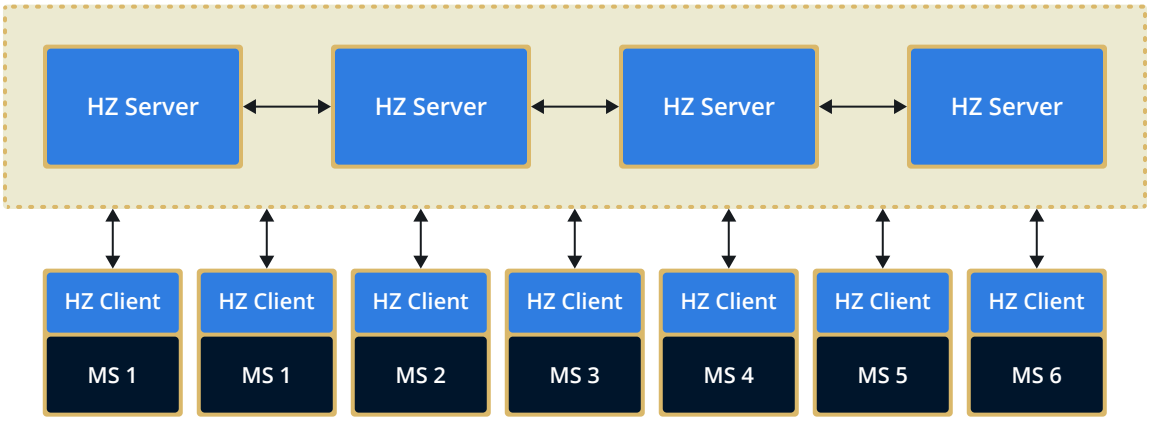

Figure 1 Microservices deployed as Hazelcast (HZ) Clients (recommended)

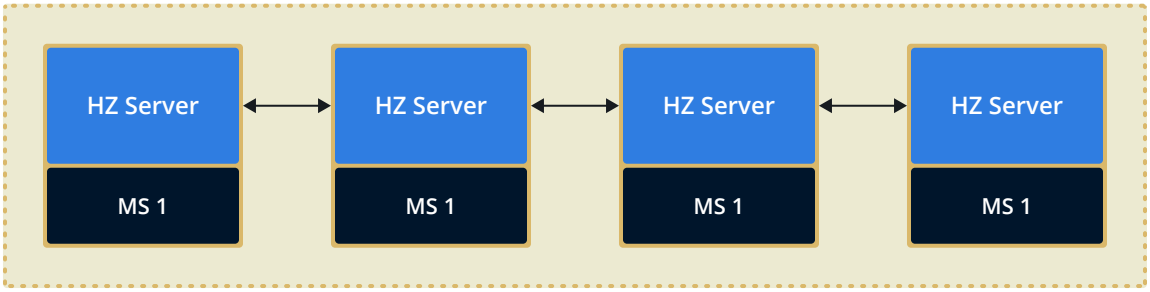

Figure 2 Microservices deployed with embedded Hazelcast (HZ) Server

Furthermore, separating the Hazelcast server nodes from the Microservices allows the Hazelcast cluster to be upgraded without having to stop all the Microservice clients (assuming the clients are coded with suitable cluster connection fault tolerance / retry logic). In our platform, we have far more services and instances of services than we do Hazelcast server nodes.

2.2 DISCOVERY

Out of the box, Hazelcast defaults to Multicast networking for node discovery and we found that developers can have clusters of Hazelcast nodes up and running in a matter of minutes. Once in production, a Microservices platform will often be used in conjunction with a PaaS or Cloud based deployment. Hazelcast currently supports the following additional networking discovery mechanisms:

- Fixed IP addresses
- Apache jclouds
- AWS
- Azure
- Consul
- etcd
- Eureka
- Kubernetes
- Zookeeper

For an up to date list, see http://hazelcast.org/plugins/?type=cloud-discovery. All of these use the Hazelcast Cloud Discovery Provider SPI and you can easily add your own.

Once Hazelcast nodes have been discovered, plain TCP/IP is used for all further communication between members and between clients and members. Supported discovery mechanisms allow Hazelcast to fully leverage many cloud environments such as OpenShift, Cloud Foundry or Docker containers.

While you may want to control all aspects of your cloud deployment, ready-made templates for Hazelcast are in the AWS Marketplace and Azure Marketplace. Hazelcast also supports Docker and has images on the Docker Hub[2].

2.3 SOLVING COMMON MICROSERVICE NEEDS WITH HAZELCAST

2.3.1 MULTI-LANGUAGE MICROSERVICES

Hazelcast has clients for several languages (currently Java, C#, C/C++, Python, Node.js and Scala) and support for a wide variety of serialization choices and the ability to plug in your own. So even though the Hazelcast server nodes are run on Java, the Microservices which are typically Hazelcast clients. can be written in any common language. Since the Hazelcast Open Client Protocol[3] is open-sourced and documented, clients for other languages are able to be built as well.

The wide programming language support also opens up the option of writing services in different languages from each other, although this flexibility becomes a business/commercial decision rather than a technical one. Multiple languages require multiple competencies and other factors that could make this optionality less advantageous.

2.3.2 SERVICE REGISTRY

There are two approaches to creating a Service Registry using Hazelcast, Named Cache and Named Channel.

2.3.2.1 Named Cache

If a Service Registry is needed, a simple fault tolerant and highly available mechanism could be put in place using a simple named `IMap/JCache`. As each service comes online it simply adds an entry to the distributed `IMap/JCache` with its service name, IP, port and a timestamp. Leveraging a service registry in this regard does put some additional complexity back on to each client Service since they now have to go out and contact each service.

[2] https://hub.docker.com/r/hazelcast/hazelcast/
[3] http://docs.hazelcast.org/docs/HazelcastOpenBinaryClientProtocol-Version1.0-Final.pdf

2.3.2.2 Named Channel

As each Microservice knows if it needs other services to perform its actions, we used Hazelcast to send a request to a named channel (`IQueue`) and wait for a response. Either the request is serviced and responded to in a timely manner (say 2 seconds), or the caller moves on and handles the unavailability accordingly. Obviously without a messaging/event backbone, each node would need to register itself somewhere so that other nodes could locate it. This would add unwanted complexity to the system.

2.4 COMPLEXITY AND ISOLATION

Microservices architectures are inherently more complex. Monolithic apps are far easier to develop and debug (when viewing the platform as a whole). Having Inter-Process Communication (IPC) just makes things harder – there's no way around that. Now the system and services have to deal with network communications, failures, rebalances, splits etc. It's much harder to debug and test the entire system. We have found that it is significantly harder triaging and diagnosing defects.

Hazelcast's robust networking and partitioning made this easier and allowed us to focus on our service implementations. As a side note, prior to using Hazelcast, we used the fairly common JMS+Spring setup. We used that for two weeks, switched to Hazelcast, and never looked back.

One benefit of a Microservices platform is that individual developers do not need to understand or test the entire system in order to add value to a team. Furthermore, technology risk is contained within a much smaller piece of the larger system. This allows for prototyping or proof-of-concepts to be tested in isolation. The worst case scenario is that if a piece of the overall system isn't working as well as expected, or becomes hard to maintain, then it can be easily refactored/replaced with a minimal amount of risk to the platform.

2.4.1 DATA STORAGE AND ISOLATION

One of the challenges of using Microservices is data sharing. Ideally, in order to keep your services isolated and independent, data should never be shared via a repository – only via service requests. Alas, real-world scenarios often get in the way of this: such things as centralized databases do actually exist in the wild. As long as the team remains acutely aware of this need for data isolation, they should be able to utilize a shared data store as long as one service does not depend on the schema or data directly retrieved from that data store.

If a single service needs to persist data, it should be able to do whatever it wants to the structure and content of its own data whenever it needs to. Care should be taken when running multiple instances of the same service (or different versions thereof) that operate on the same persisted data.

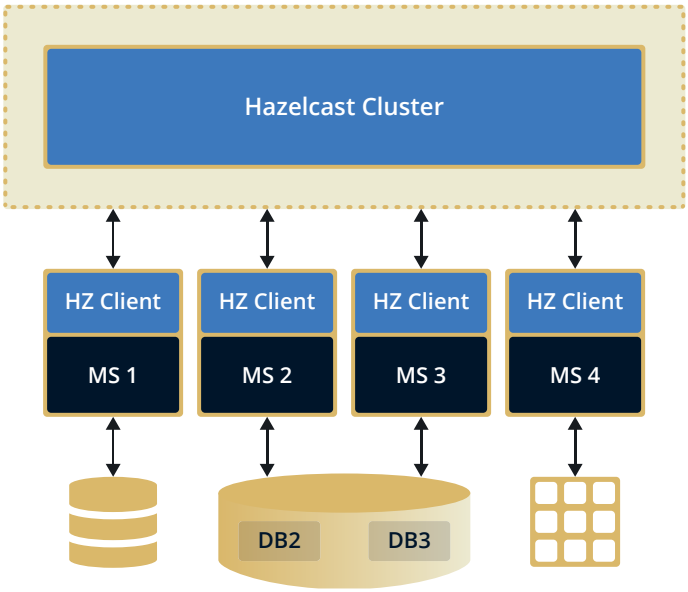

Figure 3 Separate and isolated data store per Service

When we started using Hazelcast as the backbone for our Microservices platform, we had a single database. Splitting the monolithic application also meant splitting the monolithic database. Gone were the safety nets of foreign keys, but it did force us to ask which service owned the data, and why other services felt that they needed access to it. As we built more services and functionality we started leveraging Hazelcast for its data storage abilities in lieu of data stores (Relational and Graph). We then realized that it was easy for other services to inspect or rely on the contents of any named `IMap`.

We had two options at that point:

a. Spin up a separate Hazelcast cluster for each service that needed in-memory data storage.
b. Treat the single Hazelcast cluster the same way we treated a shared RDBMS and trust the development team to follow one of the few rules on the project.

We chose (b) because it was simpler for us to manage and maintain. Furthermore, by leveraging Hazelcast as the control-plane for our platform, we can inspect, monitor, install and control the services from within a single cluster via the admin channel on our DevOps UI. This made using separate named `IMaps` (prefixed by service name and version) the logical choice for us.

2.4.2 SECURITY

The choice of where your distributed platform is hosted (in-house data center vs external data center vs external host) and the type of data you are hosting should impact your thoughts on security. The more network interconnections your platform has, and the more distributed the data, the larger the potential attack surface is. For Vertex's lending platform, which stores financial data, we leveraged cloud services for hosting. Since we don't trust anyone with our client's data, encryption of in-flight and at-rest data was paramount.

Hazelcast has the ability to plug-in various security mechanisms to secure the in-flight data transmissions as well as controlling connections to the cluster (both client and server nodes). Encryption of at-rest data, both live and in backups, was readily handled by the storage platforms themselves or by passing sensitive data through encryption libraries prior to storage.

2.4.3 SERVICE DISCOVERY

A Service knows ahead of time that it needs or may need something from another service, so it just makes the request at runtime. If another Service can answer in a reasonable time frame, that Service will do so. If no response can be given within a reasonable time frame, then the call fails and the caller has to handle the failure as it sees fit.

Being able to find out what other services are running on the platform serves little business benefit to an instance of a Service – at least in our experience leveraging a message oriented architecture. Our Admin channel/control-plane has a mechanism to discover what's running and where (IP, ports, service name, times, metrics etc.) but that's not leveraged by the services themselves, only for DevOps. For a more disconnected approach leveraging HTTP/REST, a Service Discovery process is required so that one node can contact the HTTP endpoint of another in order to make a request.

2.4.4 INTER-PROCESS COMMUNICATION

A Microservices architecture doesn't dictate the communication mechanism. While HTTP/REST is occasionally used, leveraging a messaging system such as that provided by Hazelcast has some significant benefits:

- The ability to defer processing of certain messages until later can be great for live service deployments – when a service is taken offline for maintenance / upgrade, the data isn't lost.
- If the processing service knows that some of the callers may have already moved on with life, the service can simply drop any expired requests when it comes back online thus saving processing those requests and allowing the service to catch up quickly (this is something Vertex built in to our communication layer).
- The ability to have simple load balancing by bringing up multiple instances of the same service and have them listen on a single shared queue of requests – there's no need for an additional load balancer process/box in this scenario which makes the platform simpler. For most services, being able to bring up multiple independent copies of a service and have them agnostic about their peers makes load balancing and redundancy really easy.
- A shared request queue approach also facilitates fault tolerance – if one instance of a service crashes, the request will ultimately time out and the caller will be notified, but meanwhile, the other instances of the same service can continue processing requests.

2.4.5 EVENT STORE

Having a messaging system in-place also facilitates an event store / event bus model which can greatly aid in distributed fault diagnosis as event logs can be captured and replayed in development. Hazelcast's communication mechanisms allow event stores to be created with relative ease.

I was first introduced to this concept on a Foreign Exchange trading desk several years ago. Initially, I didn't fully appreciate the enormous benefits of this ability, but once we had a report from the desk that "something odd happened around 11AM EST" we realized that we could replay the event logs and everything that had happened (all price ticks, orders, confirmations etc.) from the morning directly in our IDE at varying playback speeds (like running a tape recorder in fast-forward mode) and hey, presto – we also saw something odd. This is an incredibly powerful mechanism, and we leveraged it repeatedly on that engagement.

Since events are immutable, Event Stores are like accounting systems – you never modify entries, only enter correcting/modifying transactions. One of the drawbacks of distributed systems is that 2-phase commit transactions are rarely used due to the complexity and performance of running a transaction across process boundaries. Being able to leverage an event store allows similar distributed transaction behaviors but with an acceptable performance (rollbacks are simply reversing entries of previously recorded events). Having an immutable chronological entry of all events that impact a system also enables replays of the system at varying speeds – extremely useful for reproducing defects as well as load and regression testing a system.

2.4.6 COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS)

At its core, CQRS is simply about separating the writing of data from the reading of it. In many scenarios, the nature and timing of read and writes of persisted data is highly differentiated. Optimizing a storage system for reading of data often impacts write performance (and vice-versa). Separating the write store from the read store separates those concerns and allows each to be optimized and leveraged accordingly. CQRS coupled with an Event Store allows for significant flexibility in querying – both in terms of the types of queries and performance profiles. While it is additional work and overhead to separate and maintain the concerns when compared to traditional data stores/services, the benefits can be significant.

Queries for data by Services are simply requests placed on to the Hazelcast messaging bus. Responses to data queries can be provided by any number of Services, each with their own level of aggregation, filtering criteria and performance. This allows for significantly different querying profiles for different Services on the same dataset simply by adding an additional query service to the platform. If a new Service requires data aggregated at a level that isn't already available in the platform, then either an existing service can be modified to supply such data, or an entirely separate one can be built and deployed specifically designed for the new requirements. In some scenarios it maybe non-trivial to modify an existing service and the lowest barrier to delivery (and thus risk) may well be to deploy a new query service.

3. Conclusion

Leveraging Hazelcast for our Microservices platform enabled us to focus on the business logic and the tasks of solving our client's problems rather than on the infrastructure and development of our network communication. Once Hazelcast was in place, we were able to rely on it and build additional services and functionality on top. The continued development, performance improvements and active user support make it our continued go-to platform for network distribution and in-memory data storage.

hazelcast

350 Cambridge Ave, Suite 50, Palo Alto, CA 94306 USA
Email: sales@hazelcast.com **Phone:** +1 (650) 521-5453
Visit us at www.hazelcast.com