High Availability in Griddable.io

Executive summary

Griddable.io is an industry-leading timeline-consistent distributed data synchronization grid across a range of source and target data systems. Griddable.io's flexible shared-nothing architecture with strong consistency guarantees allows for building highly available deployments. Thus, Griddable.io is suitable for providing solutions with low-latency, reliable data synchronization for many mission-critical use cases inside a modern enterprise. This whitepaper describes possible approaches for achieving high levels of availability that can fit the needs of a wide range of enterprises.

Introduction

Griddable.io is a SaaS company that delivers a platform for data synchronization & integration across enterprises and clouds. Griddable.io provides:

- Source-independent transactional grid optimized for change data based replication
- Policy engine for selective replication, on-the-fly data transformation and conflict resolution
- Pluggable architecture to extend support for data sources, consumers, and policies
- A set of tools and web-based UI to simplify management in a geo-distributed environment

Support for running high-availability and reliable distributed applications was one of the key design aspects when designing Griddable.io. This whitepaper studies and presents several approaches to ensure the high availability of data synchronization solutions build using Griddable.io's technology. First, this paper defines the problem statement precisely. Next, it provides some background information to serve as the basis for the detailed study of the problem. Then, it discusses how to ensure high availability within a single data center/availability zone and across data centers, availability zones and regions. This paper also looks at the availability problem end-to-end, i.e. how data flows through the Griddable.io transaction grid. Finally, it reviews and summarizes this information.

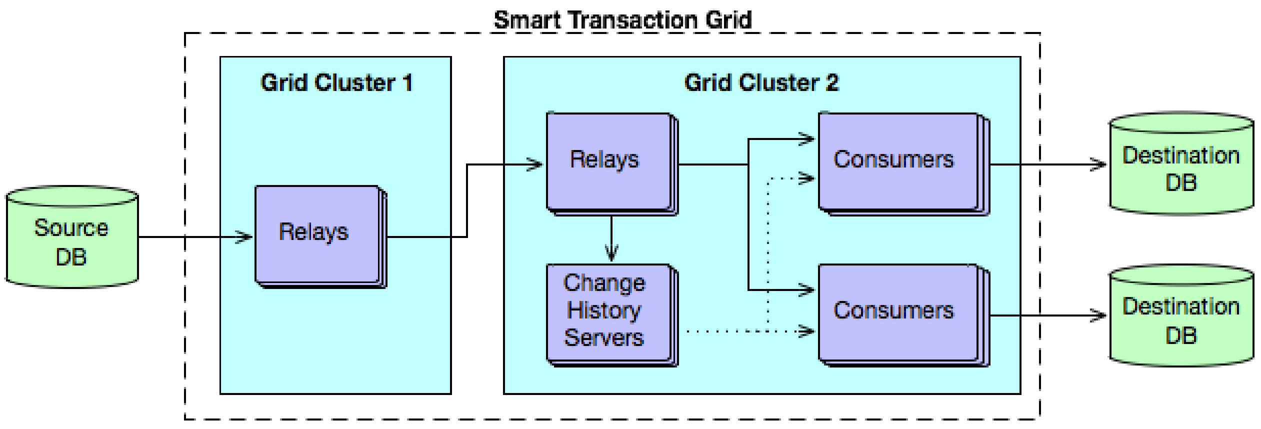

Figure 1: High-level Griddable.io architecture

Problem statement

In a typical Griddable.io deployment there are four main types of systems and services:

- Upstream source database system
- Infrastructure services
- Consumer services
- Downstream systems like other database systems, Hadoop and other file systems and message queues

All four types of components affect the availability of the system. This paper focuses only on HA for the Griddable.io infrastructure components and the consumer services. There is an already existing wide range of available resources for configuring and maintaining HA for database systems, Hadoop and message queue systems.

This paper uses standard terminology from distributed cloud computing such as availability zones and regions.

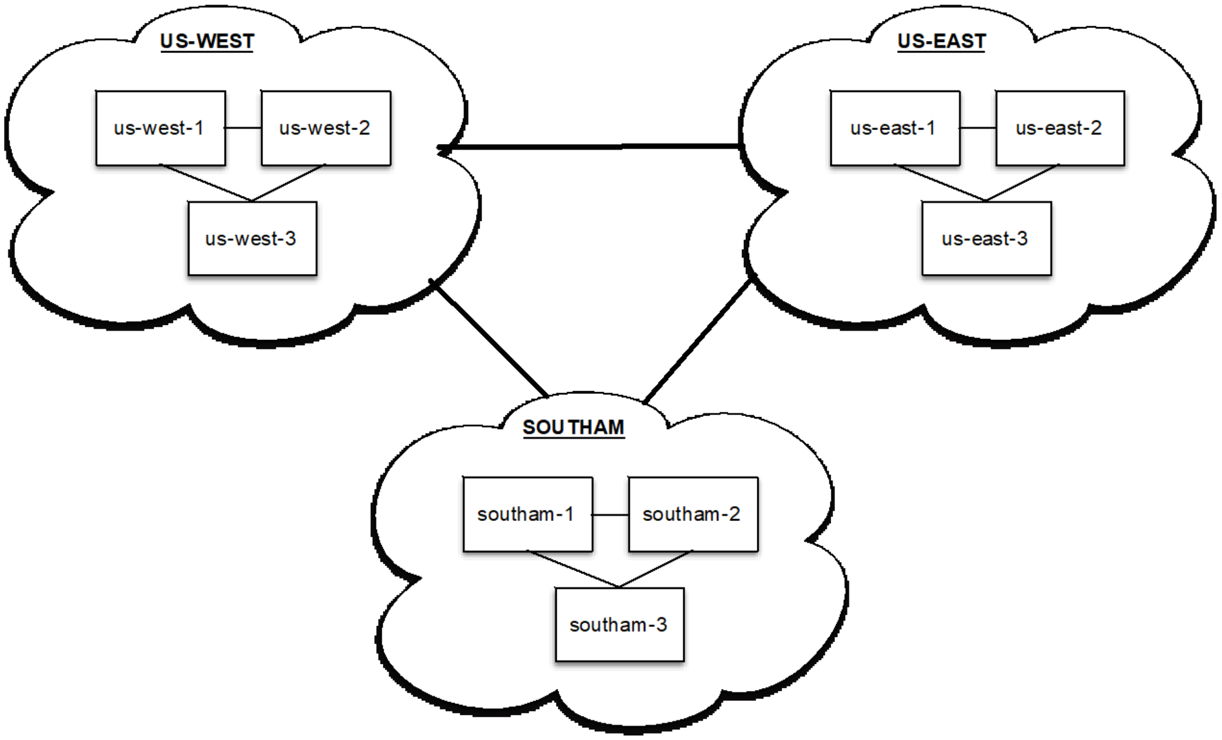

Figure 2: Typical cloud deployment model

For example, in Figure 2 there are three regions (US-WEST, US-EAST, SOUTHAM) each with three availability zones. Each availability zone (AZ) provides isolated power and networking which provide a high-level of isolation from the other AZ in the same region. The AZ within the same region typically provide high level of network connectivity with low latencies due to the close geographical proximity. Regions are situated across the globe thus providing the highest level of isolation but the network connectivity is more limited due to the long geographical distances.

Popular public cloud vendors like Amazon Web Services, Google Cloud Engine and Microsoft Azure all provide a similar model to the above although naming, geographies and topology may differ.

Design principles

Timeline consistency

One of the most important design principles in Griddable.io is the timeline consistency for all components on the grid. This means that these components (including the geographically distributed in different AZs) follow the same commit timeline as defined by the source database. This includes all the updates that happened in that database in exactly the same order. The logical clock associated with the commit timeline is also preserved.

The state (aka the checkpoint) of each component (relays, change history servers, consumers) is determined by the last consumed transaction in that the commit timeline. Further, Griddable.io middleware components such as the relays and change history servers also have a low watermark of the earliest transaction that they can serve.

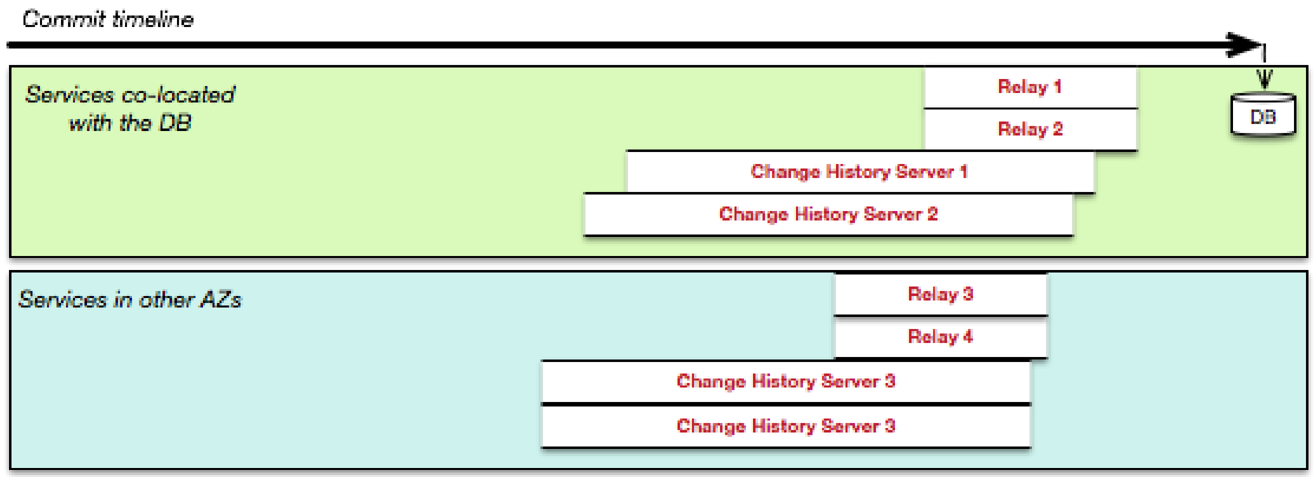

Figure 3: Timeline consistency

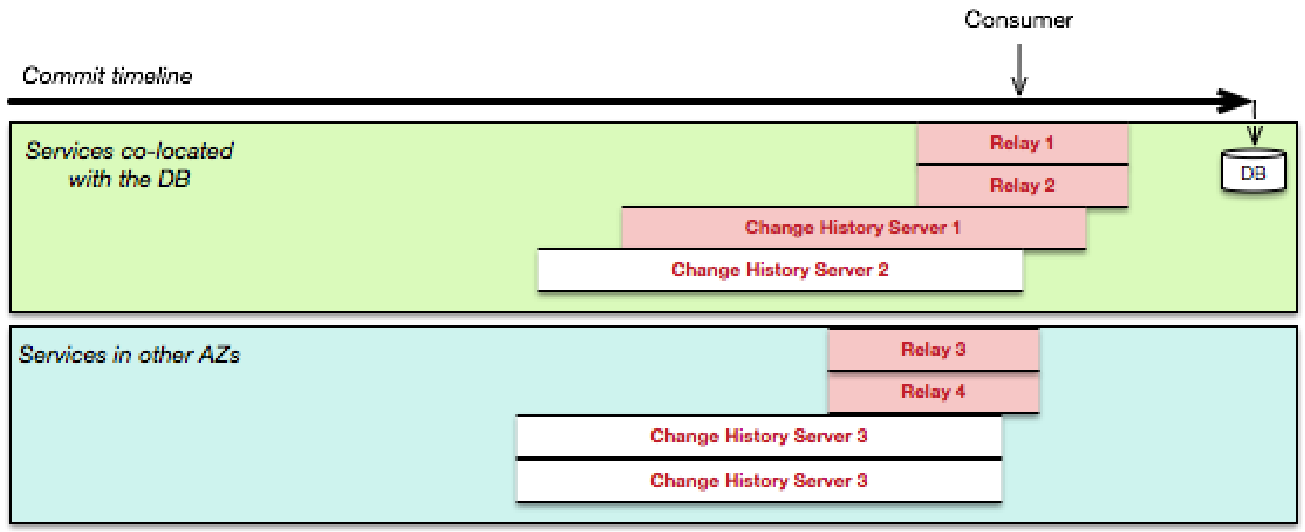

Figure 4: Serving in ACDS as cache hierarchy

Figure 3 illustrates how Griddable.io components form a cache hierarchy. A consumer can potentially be served by any component whose range contains the consumer checkpoint. Figure 4 highlights the components that can serve the consumer.

Such flexibility and a wide range of alternatives that can be used to serve the consumer promote a very high level of availability.

Notes

The Griddable.io client library that manages the connections the relays and change history servers on behalf of the consumer services prioritizes connecting to relays over change history servers as the former generally provide better latency due to the fast in-memory processing path

Pull-based architecture

Another critical design principle is the pull-based architecture. In Figure 1, all connections between all systems (DB, relays, change history servers, consumers) are pull-based with connections initiated by the downstream components.

This architecture has the following implications.

- The consumption state (the checkpoint) is localized and managed by the consumption side of each connection. There is no need for centralized repository for managing state which, if used, can become a single point of failure (SPOF) for the entire system.
- Unavailability of the consumption side does not affect the availability of the producer side. The latter just continues processing. Thus, unlike push-based architectures, the producer side does not need to decide between (a) stopping processing until the consumer side becomes available potentially making the producer side unavailable too or (b) moving on and sacrificing consistency and correctness.
- Unavailability of the producer side does not affect the availability of the consumer side. If the former becomes unavailable, the latter can failover to a different instance or cluster (due to the timeline consistency).

High availability within a single availability zone

Within an availability zone, there is generally a high level of network connectivity, high bandwidth and low network latencies. This facilitates many deployment topologies to ensure the high availability of Griddable.io-based data synchronization solutions.

Relays

The primary approach for maintaining high availability within an availability zone for the relays is through clustering and intra-cluster/inter-cluster replication. Consumers, change history servers and other relays can be configured to consume from a cluster of relays. As discussed above, the timeline consistency property maintained by each relay ensures that all relay instances are fully interchangeable from the point of view of the downstream consuming service.

On start-up, the downstream service will choose an instance to connect to. Typically, this is done by choosing a random relay instance from the configured ones. If that instance is or becomes unavailable at any point, an automatic fail-over to a different instance will occur. More sophisticated load-balancing policies will be added with the Helix clustering feature (see below).

This allows Griddable.io customers to build truly shared-nothing clusters with high availability and scalability.

Relay clustering

There are three general types of relay clustering alternatives:

- Customer-defined clustering with a load balancer
- Clustering through a Griddable.io-managed cluster configuration
- Clustering using Helix

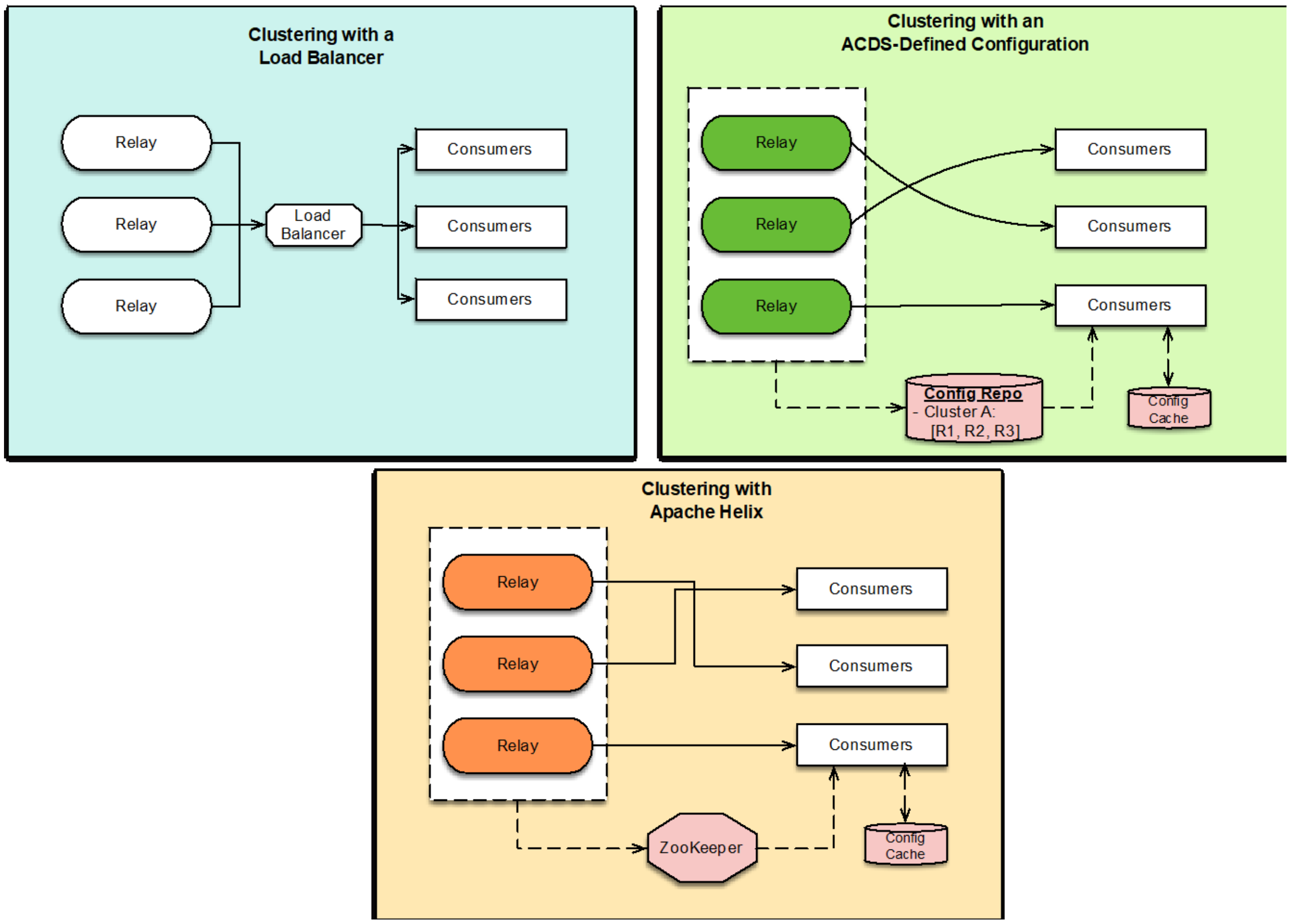

Figure 5: Relay clustering types

Note that consumers do not rely on the clustering approach to determine relay availability. Clustering bears importance only on relay instance discovery. Each consumer independently tracks which relay instances are available. There are three alternatives.

User-defined clustering with a load balancer

Consumers use standard HTTP to communicate with the relays. Thus, the relay instances can be placed behind a load balancer or a reverse HTTP proxy. The management of these instances is largely done outside of Griddable.io. The load-balancer/proxy is assigned a single DNS hostname that is used by consumers to access the relay servers.

This approach works well if the customer already has existing infrastructure that utilizes load balancers and the customer wants to capitalize on the existing experience operating it.

There are number of alternatives for load-balancers. These can generally be grouped into three categories:

- Software load-balancers like HAProxy[1] and NGINX Plus[2]
- Hardware load-balancers like Citrix NetScaler[3], F5 Big-IP[4], and others
- DNS round-robin

Each of these alternatives have their pros and cons, a discussion of which is beyond the scope of this whitepaper.

The main downside of this approach is that the load-balancer becomes a SPOF of the entire cluster. This runs counter to Griddable.io's approach of fully-distributed, shared-nothing architecture. It is expected that customers choosing this approach have established ways to maintain such availability in the load balancer. Otherwise, Griddable.io recommends the Griddable.io-managed cluster configuration approach presented next.

Notes

1. The load-balancer should be configured to maintain persistent HTTP connections between the relays and the consumers. Currently, consumers use continuous polling for more data. It is important that all those requests are routed to the same relay instance for a couple of reasons. First, there is a hand-shake between the relay instance and consumer at the beginning of a session. If subsequent requests are re-routed to a different instance, this can lead to the consumer being unable to process the response if there is a difference in relay configurations. Second, persistent connections ensure that the overhead of maintaining these HTTP connections on the relay side is kept to a minimum.
2. For monitoring of relay availability from load-balancer, relay instances provide the `/admin` HTTP interface. An HTTP `200 OK` response indicates that the relay is up and available. An HTTP `500 Internal Server Error` or `503 Service Unavailable` indicate that the relay service should be considered unavailable and consumer requests should be redirected to other instances

When using this approach, configuring the consumers is as simple as setting the relay hostname.

```
grid consumer connect --relays acds-relays.company.com
     –-name myconsumer
```

Clustering through a Griddable.io-managed cluster configuration

The definition and management of relay clusters through Griddable.io is the currently recommended approach for most deployment scenarios.

The idea is to use the tools and UI provided by Griddable.io to define and distribute the configuration about the available cluster (see the second diagram on Figure 5). That information can be edited and

[1] http://www.haproxy.org/
[2] https://www.nginx.com/products/nginx/
[3] https://www.citrix.com/products/netscaler-adc/
[4] https://f5.com/products/big-ip/local-traffic-manager-ltm

stored a in configuration repository which can be a local file system, on shared storage as an NFS mount or through the Griddable.io Management Server.

First, this approach does not introduce a potential single points of failure (SPOF) on the critical data synchronization path. The cluster configuration information is sent asynchronously to all components which cache it. They will be able to continue to operate even in the case of temporary unavailability of the configuration repository. Outage can occur only if there is a simultaneous failure of the configuration source and all instances of the required Griddable.io replication grid.

Second, there is a tight integration of this approach with all Griddable.io tools and the UI which ensures a more consistent user experience.

This approach can still be integrated with external monitoring tools for availability through the aforementioned `/admin` HTTP call or through the metrics exposed by Griddable.io. Please refer to the Griddable.io documentation about the available metrics.

To define a cluster named “clusterA” consisting of two relay instances “relay1.company.com” and “relay2.company.com” one can use the Griddable.io command-line tool:

```
grid service-cluster create --name clusterA
     --relays relay1.company.com,relay2.company.com
```

To configure the consumer instance “myconsumer” to read from the relay cluster “clusterA”, one can use:

```
grid consumer connect –-name myconsumer
     –-service-cluster clusterA
```

Subsequent changes can be made to the relay instance in the cluster. For example, adding or removing relay instances can be done through:

```
grid service-cluster add --name clusterA
     --relays relay3.company.com

grid service-cluster rm --name clusterA
     --relays relay1.company.com
```

Clustering using Helix

This feature is coming soon in a future release. This extends the Griddable.io-managed cluster configuration alternative with integration with the Apache Helix[5] cluster management framework.

[5] http://helix.apache.org/

This approach provides the ability to manage the clusters much more actively. It allows:

- Finer granularity tracking of the state of each instance.
- Ability to easily setup typical cluster configurations like Leader/Standby and Master/Slave.
- Better tracking of instances that have experienced an error and marking them unavailable automatically.
- Ability to quickly add and remove instances to clusters
- Ability to take instances out of rotation without taking them down (e.g. for debugging or maintenance)
- Ability to define sophisticated load-balancing algorithms

Chaining and tiering

Chaining is the use of one relay instance or a cluster as a data source for another relay instance or cluster.

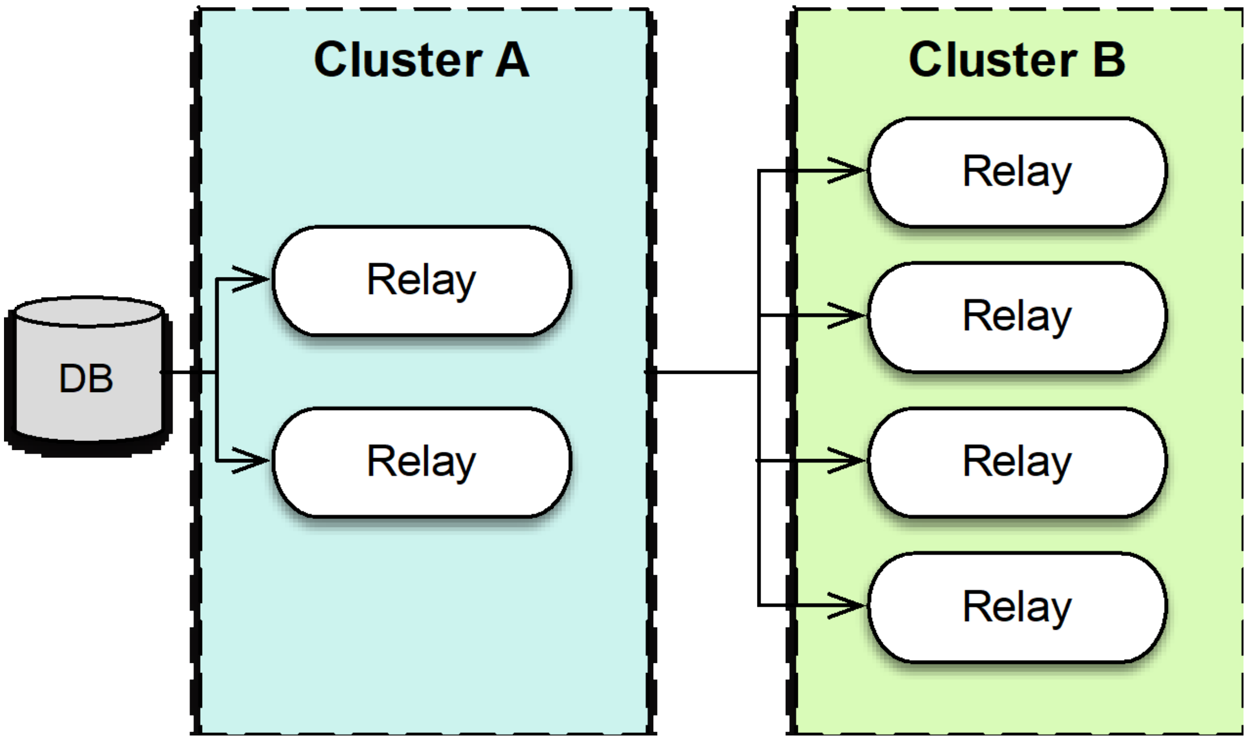

Figure 6: Relay chaining

Relay chaining has application in both same-AZ deployment and cross-AZ deployments (discussed in the section "High availability across availability zones").

Intra-AZ chaining allows to (a) grow relay clusters to a practically unlimited number of relay instances and consumers and (b) define different levels of service based on tiering of the relays.

For the first use-case, consider Figure 6, where 6 relays are available to downstream consumers while only having two relays directly connected to the database. This is important because relays generate some small overhead in the source database. That overhead may add up if this is a high-traffic DB and there are multiple relays connected to it. On the other hand, relays are optimized to serve many streaming consumers. Thus, Cluster A can serve a much higher number of relays in Cluster B. Then, customers have choice to configure downstream consumers to connect to the relays in Cluster B or both clusters. This architecture allows serving of practically unlimited number of consumers.

The second use case is an extension of the first one. Consider Figure 7, which switches the names of the clusters to Tier1 and Tier2 to denote their purpose. Tier 1 is closer to the database and thus provides lower latencies. On the other hand, it has more limited capacity and disruptions to the service have harsher implications. Thus, solution designers can reserve this cluster to a limited number of consumers which require higher Quality-of-Service (QoS) in terms of latency and predictability. In the current release, each relay in a chain can be configured separately. Future release will allow the central management server to hold these configurations and push them automatically to the relays.

On the other hand, Hadoop consumers tend to be more numerous and do not have as strict latency requirements. Sometimes they come in big bursts due to (misconfigured) batch jobs. Therefore, they are isolated to a separate (chained) cluster with lower QoS.

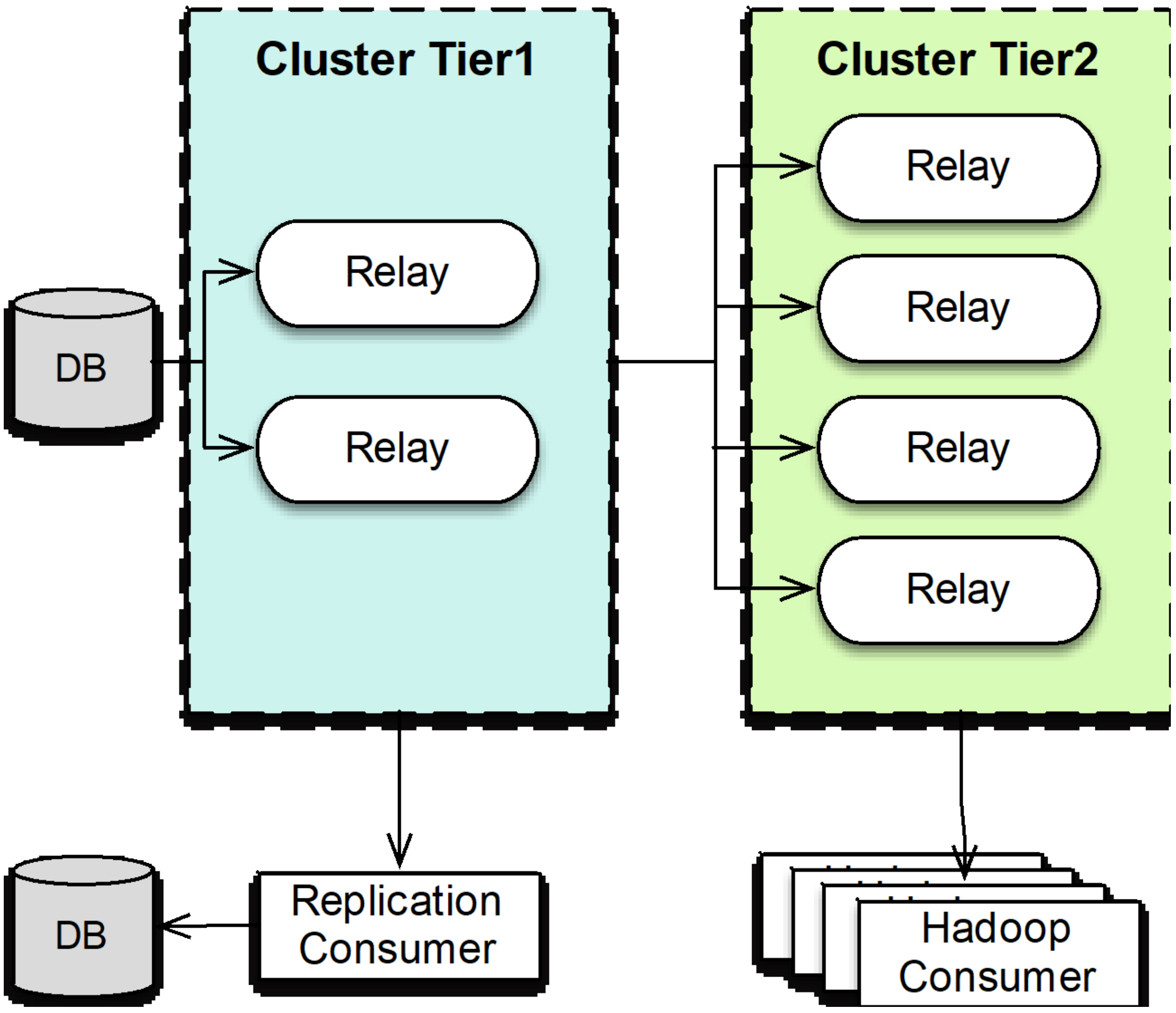

Figure 7: Relay tiering

This approach can be extended to have several specialized clusters for different classes of consumers if necessary. This can be combined with policies. For example, the Tier2 can enforce a policy that all sensitive fields are removed so it does not get exposed to less secure analytics jobs.

Change History Servers

The change history service has two parts: the Change History Consumer which is responsible for maintaining the persistent log and the Change History Server which is responsible for serving consumer request from the persisted log.

The change History Server

The Change History Server is like the relays and its clustering types are like relay clustering ones: with a load-balancer, cluster configuration, and Helix-based. The main difference is that chaining of change history servers is currently not supported.

The commands for managing the change history server clusters are also similar. To create a new change history cluster, one can type:

```
grid service-cluster create --name clusterX
     --chs chs1.company.com,chs2.company.com
```

One can also combine a group of relays and change history servers into a single cluster:

```
grid service-cluster create --name clusterY
     --chs chs1.company.com,chs2.company.com
     --relays relay1.company.com,relay2.company.com
```

The Change History Consumer

The Change History Consumer is the other part of the change history service and it is managed similarly to any other consumers. This is described in the next section on consumers.

Consumers

Consumer high availability is not managed by Griddable.io in its first release. A future release will add support for automatic consumer failover through the Helix cluster management framework (see the "Clustering using Helix" section above).

The recommended approach for consumer process HA currently is to use a container restart policy through Docker or container management framework such as Kubernetes. On restart, the consumer will re-read the last persisted checkpoint and continue from that point on.

High availability across availability zones

This section reviews an HA deployment spanning multiple AZs in the same or different regions. The recommended approach is to use chained relay cluster as show on Figure 8 Cross AZ Relay Chaining .

This approach limits the amount of cross-AZ network traffic, as it happens only between the pairs of relay clusters. Any additional change history services and consumers are connected only to their local cluster.

Single node failures within the local relay cluster are handled in the same as the single AZ HA scenarios. Consumer services will automatically switch to other available relays.

If the entire local relay cluster fails (say cluster3 with R31 and R32), consumers can be configured to switch to a different AZ (e.g. cluster2). Even though this will most likely increase the cross-AZ network traffic, it is a temporary solution until the relay cluster is restored.

```
grid consumer connect --name myconsumer
     --service-cluster cluster2
```

Currently, the re-configuration needs to be performed manually. In a future release, with the addition of Helix clustering support, this can happen automatically. Consumer services can be configured to access both clusters. There will be weights associated with those connections and the client library will always prefer to connect to the available cluster with the lowest weight.

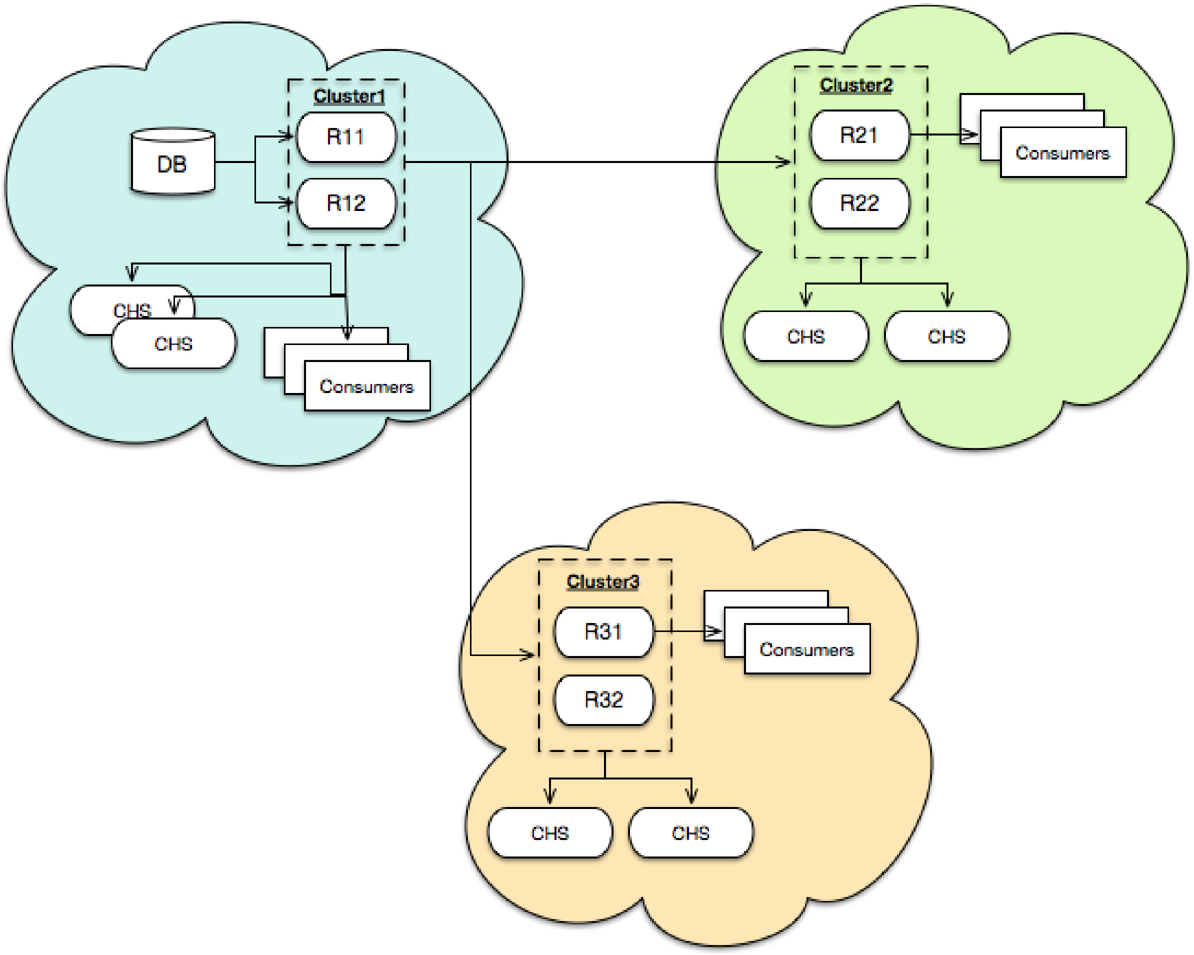

Figure 8: Cross-AZ relay chaining

If the network connectivity between two AZ is interrupted, an AZ fail-over needs to be performed. Like the local relay cluster failure, the relay cluster needs to be re-configured to connect to a relay cluster from a different AZ.

Data availability

A lot of the previous discussion focused on availability at the system components level. That is a necessary condition for the end-to-end data synchronization flow to be considered available but it is not sufficient. Even when components appear available they may not be able to replicate data due to internal processing errors or upstream issues. Detection of such issues becomes increasingly important in highly distributed systems as local components may look healthy due to limited view of the global state of the system.

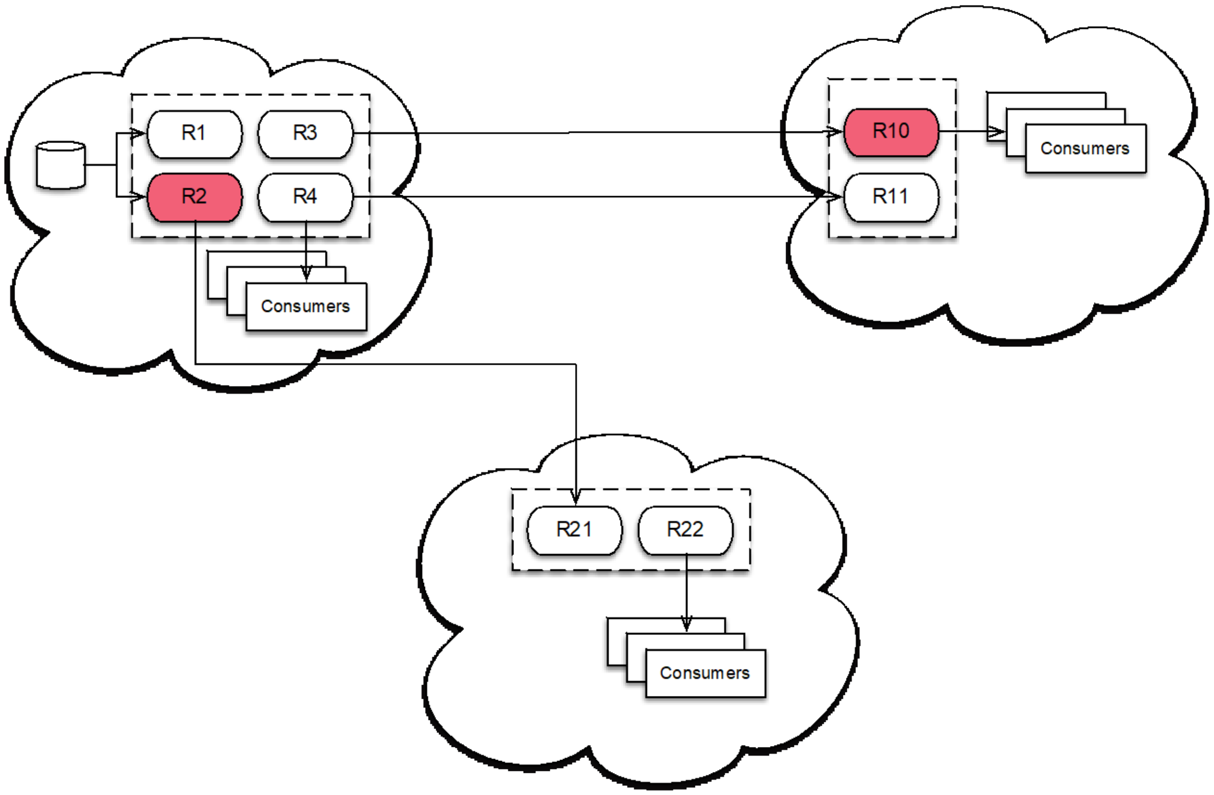

Figure 9: Sample failure scenarios

In Figure 9, R2 is experiencing a GC storm due to misconfiguration and is unresponsive or returns frequent Out-Of-Memory errors. R10 is experiencing a network connectivity issue to R3. In both cases, this may lead to data unavailability in downstream consumers to R22 and R10 since those may appear healthy but they are unable to serve change events.

Griddable.io provides several safeguards against above conditions.

First, the change history servers (apart from serving lagging/misbehaving consumers) can also act to ensure data availability in case relay cluster experiences issues with drops in its event buffer caches retention.

Second, any error in communication between two components (e.g. consumer and relay or change history server) will trigger a fail-over to a different instance.

Third, if the consumer detects a period during which no changes have arrived from a relay, it will also trigger an automatic failover to a different instance.

Finally, all components (relays, change history servers, consumers) expose metrics about the last processed transaction and the overall replication lag. This can be used along with external monitoring systems to trigger alerts and operator intervention to maintain the required level of availability. The metrics collected from each of relays and consumers can be fed to any logging/monitoring system (like ELK) and anomalies can be detected.

Conclusion

This paper describes the HA support in Griddable.io's transactional replication grid. It shows that Griddable.io's pull-based timeline consistent architecture allows for building of highly-available mission-critical enterprise applications for real-time data synchronization and integration.

HA dimension	Current Automatic Support	Current Manual Support	Future
Relay (Intra AZ)	• Griddable.io-based clustering • Relay chaining • Relay tiering	• Manual LB clustering • Cluster state monitoring	• Helix-based clustering
Change history (Intra AZ)	• Griddable.io-based clustering	• Manual LB clustering • Cluster state monitoring	• Helix-based clustering
Consumer (Intra AZ)	• Container restarts	• Cluster state monitoring	• Helix-based clustering
Cross AZ		• Manual cross-AZ failover	• Cost-based load balancing • Automatic cross AZ failover
Data	• Change history service • Failover on no incoming data • Failover on error	• Watermark monitoring	• Metrics-based failover

Table 1: Summary of HA features

griddable.io