Building Microservices With Redis

WRITTEN BY SHABIH SYED SR. DIRECTOR OF PRODUCT MARKETING AT REDIS LABS
AND MARTIN FORSTNER SOLUTION ARCHITECT AT REDIS LABS

CONTENTS

According to Gartner, the worldwide enterprise application software market is growing at an annual rate of 10.4 percent and is expected to reach $310.2 billion by 2022. These applications need to deliver high-speed data processing while simultaneously depending on a variety of adaptable components to accommodate business requirements. To address these demands, many companies are turning to rapidly deployable microservices, which offer greater flexibility and agility than monolithic architectures. However, with a microservices architecture, you have to pay significant attention to how you share data between microservices and how you handle inter-services communication at scale.

Redis Enterprise, which is built over open-source Redis, offers a high-speed, multi-model database with high availability and durability options that are crucial when building a reliable event store for trading messages between your microservices. Redis Enterprise is easy to operate, and it's available as a cloud service on all popular public cloud platforms and in VPC/on-premises environments. It's also available as an orchestration container using Kubernetes, BOSH, and Docker Swarm. Redis Enterprise's multi-tenant architecture can run up to a few hundred databases on a simple three-server cluster, maximizing the use of your resources.

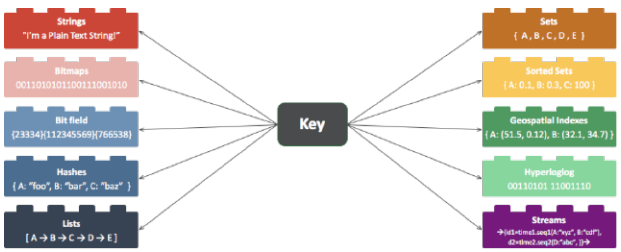

Figure 1: Data structures — Redis' building blocks

Introduction to Microservices

Microservices are in the spotlight as infrastructure building blocks because they offer several advantages that facilitate faster time-to-market for new applications or updates. These benefits include services decoupling, data store autonomy, and miniaturized development and testing. The availability of containers and their orchestration tools has also contributed to increasing microservices adoption. A recent survey of development professionals found that 86 percent expect microservices to be the default architecture within five years, with 60 percent already having microservices in pilot or production.

Microservices architectures reject the notion of a monolithic application framework that uses a single large database, instead favoring data sharing between services. Microservices embrace autonomous,

Redis Labs Named a Leader

The Forrester Wave™: Big Data NoSQL, Q1 2019

DOWNLOAD

redislabs
HOME OF REDIS

specialized service components, each with the freedom to use its own data store. In general, software applications are easier to build and maintain when broken down into loosely coupled, self-contained, logical business services that work together. Each of these smaller services (i.e. microservices) manages its own technology stack that you can develop and deploy independently of other services. You can also update and scale these highly specialized microservices independently. This architecture design is powerful because it gives your organization the ability to more efficiently allocate resources and release new applications or updates.

Performance Requirements of Microservices

With microservices, it's important to design every service to provide the best throughput. If one microservice becomes a bottleneck in the flow of data, then your whole system may collapse.

READ PERFORMANCE

A commonly used metric for read performance is the number of operations per second. However, in many cases, this metric is a combination of how fast you can run queries and how fast you can retrieve results. The speed of retrieving results is dependent upon how well you can organize and index data. A product catalog microservice, for example, may run queries that apply multiple parameters such as product category, price, user rating, etc. The database that you choose for such a microservice must first allow you to organize the data to run your queries faster, and then be able to accommodate the number of operations-per-second requirement as well.

WRITE PERFORMANCE

The easy metric here is to determine the number of write operations your microservice performs per second. Disk or network-storage-based databases are inherently slow compared to in-memory databases. Microservices that collect and process transient data need databases that can perform thousands, if not millions, of write operations per second.

LATENCY

Microservices that deliver instant user experiences require a low-latency database. Deploying a microservice close to its database will minimize the network latency. Database architecture and its underlying storage may add additional latency. Low-latency databases are needed for microservices that perform real-time analytics (such as fraud mitigation).

RESOURCE EFFICIENCY

Microservices are lightweight by design. The database footprint must be minimal while retaining the ability to scale on demand, reflecting the design principles of microservices and their agility to the greatest extent.

PROVISIONING EFFICIENCY

Microservice components need to be available for rapid development, testing, and production. Any database service used will often be required to support the on-demand creation of hundreds of instances per second.

You can categorize your microservices by their read and write operations. The following are typical accepted numbers for operations per second:

- **Very high**: > 1 million
- **High**: 500K - 1 million
- **Moderate**: 10K - 500K
- **Low**: < 10K

For latency, the typical numbers are:

- **Low**: < 1 millisecond
- **Moderate**: 1 - 10 milliseconds
- **High**: > 10 milliseconds

Types of Data Processing in Microservices

Not all microservices process or manage data at the same stage in its lifecycle. For some microservices, the database could be the source of truth, but for others, it may just be a temporary store. To understand the data needs of your microservices better, you can broadly classify the data in the following categories based on how it is processed.

TRANSIENT DATA

Data such as events, logs, messages, and signals usually arrive at high volume and velocity. Data ingest microservices typically process this information before passing it to the appropriate destination. Such microservices require data stores that can hold the data temporarily until it can be stored. These data stores must support high-speed writes. Additional built-in capabilities to support time-series data and JSON are a plus. Since transient data is not stored anywhere else, high availability of the datastore used by your microservice is critical — this data cannot be lost.

EPHEMERAL DATA

Microservices that deliver instant user experiences often rely on a high-speed cache to store the most accessed data. A cache server is a good example of an ephemeral data store. It is a temporary data store whose sole purpose is to improve the user experience by serving information in real time. While a data store for ephemeral data does not store the master copy of the data, it must be architected to be highly available, as failures could cause user experience issues and subsequently, lost revenue. Separately, failures can also cause "cache stampede" issues as your microservices try to access monolithic-backing databases.

OPERATIONAL DATA

Information gathered from user sessions — such as user activity, shopping cart contents, clicks, likes etc. — are considered operational data. These types of data power instant, real-time analytics, and are typically used by microservices that interface directly with users. This type of data is frequently used in real time to improve user experiences or provide intelligent responses, and later aggregated to support longer-term trend analysis. Even though the data stored in the datastores for this type of processing is not a permanent proof of record, the

architecture must make its best effort to retain the data for business continuity and analytics. For this type of data, durability, consistency, and availability requirements are high.

TRANSACTIONAL DATA

Data gathered from transactions (e.g. payment processing and order processing) must be stored as a permanent record in a database. The datastores used must provide strong ACID controls and must employ cost-effective means of storage, even as volumes of transactions grow.

Synchronizing Transient Data Across Microservices

While building these distributed collections of microservices is an extremely lightweight process, they can often introduce new challenges for your development teams. As Redis is a multi-model database, it is being used by developers in a microservices-based architecture to store transient data, ephemeral data, operational data, and transaction data. But storing transient data, such as events and messages for inter-services communication, is particularly catching on.

Initially, many developers tried to plumb microservices together with point-to-point communication. They very quickly realized that this approach fails at scale and should be avoided at all costs to account for crashed services, retry logic, and significant headaches when load increases — and should be avoided. Others tried relying on the decades-old enterprise service bus (ESB) to implement a communication system. Unfortunately, they struggled with slower communication speeds, single points of failure, and high configuration and maintenance complexity. Ultimately, deploying and scaling a single ESB as a monolithic runtime was found contrary to a modern architecture approach, which requires a containerized environment with effortless relocation or replication of microservices.

Today, the current generation of inter-services communication models range from message queues (MQs) to complex Kafka-esque event topic streams that provide communication and coordination for decoupled services.

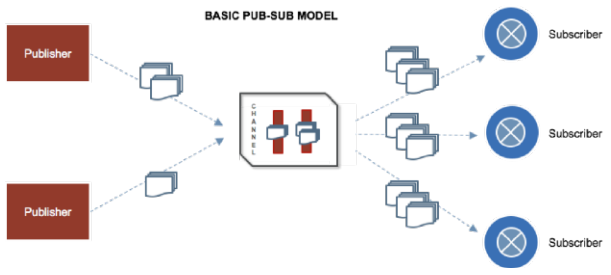

Figure 2: Basic event store model

Push/pull, pub-sub, and topic/stream event stores significantly simplify the coding of decoupled microservices, while improving performance, reliability, and scalability. This approach enables asynchronous communication in which producers and consumers of event messages interact with a queue or a stream, rather than each other. So, if a microservice dies unexpectedly, the queue or stream can still receive incoming event messages from decoupled services and store them for later, so the dead microservice will act on them when it comes back online.

Redis Streams for Interservices Communication

Microservices distribute their state information over network boundaries. To keep track of this state, these events should be stored in an event store. Since event records usually form an immutable stream of asynchronous write operations (i.e. a transaction log), the following properties apply:

1. Order is important (due to time series data)
2. Losing one event leads to a wrong state
3. The replay state is known at any given point in time
4. Write and read operations are easy and fast
5. Streams require high scalability as each service is decoupled from the others

Redis Streams, a unique new data type introduced with Redis 5.0, was built specifically to address these communication patterns. It includes a message publishing and subscription component, but the Redis Streams data structure is more abstract, which makes it ideal for event message queues and time series data.

While the existing Redis pub-sub and blocked list approach can be used as a message queue service in simple scenarios, Redis Streams provides message persistence and master/slave data replication functions. These include a new RadixTree data structure to support more efficient memory use and complex message reading. Similar to Kafka's Consumer Group, which gets data from a stream, Redis Streams serves multiple consumers. But Redis Streams also provides certain guarantees: each message is served to a different consumer, so it's not possible for the same message to be delivered to multiple consumers.

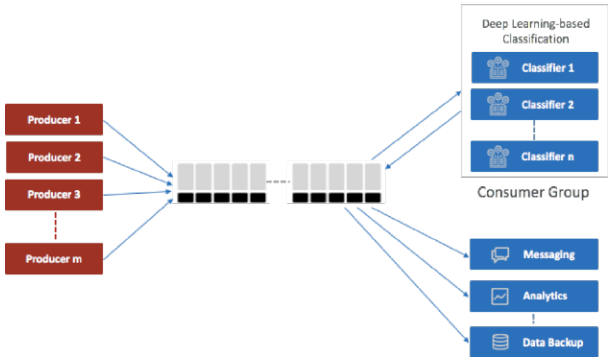

Figure 3: Redis Streams offers persistence and advanced capabilities like consumer groups

Because Redis Streams uses its primary memory for storage, it is much faster than disk-based middleware like Kafka or RocketMQ. Along

with active-active capabilities, easy and simple deployment, and fast in-memory processing, it is a must-have for managing microservices communication at scale.

Redis Streams vs. Redis Pub/Sub, Lists, and Sorted Sets

In addition to Streams, Redis offers multiple other ways to create an event-based architecture for microservices — pub/sub, lists, and sorted sets. Streams is effectively the culmination of all three methods above because it combines all their benefits while also supporting connectors for multiple clients and staying resilient to connection loss.

- Pub/sub implementation employs a simple and memory-efficient form of asynchronous communication, which requires subscribers to be active in order to receive data.
- Lists data structure, meanwhile, supports a blocking call for an asynchronous data transfer and is particularly efficient because it persists data even if a receiving microservice fails.
- Sorted sets are ideal for transferring time-series data, but unlike pub/sub and lists, they do not support asynchronous data transfers — although pub/sub, lists, and sorted sets all offer viable ways to transfer data.

OrderShop: A Sample Application With Communications Between Microservices

The OrderShop application is a simple, but common, e-commerce use case developed by Redis Labs to show how you can manage customers, inventory, and orders. It demonstrates the process of creating or deleting a customer, inventory item, or order. When any of these changes are triggered, the application asynchronously communicates an event to the CRM service, which manages OrderShop's interactions with current and potential customers. Like many common applications, the CRM service in the "OrderShop" application may start or stop during runtime so when this happens it happens without any impact to other microservices, necessitating that all messages sent to it during its downtime are captured for processing.

The following diagram shows the interconnectivity of seven decoupled microservices that use event stores built with Redis Streams for inter-services communication. They do this by listening to newly created events on a specific stream in the event store, i.e. a Redis instance.

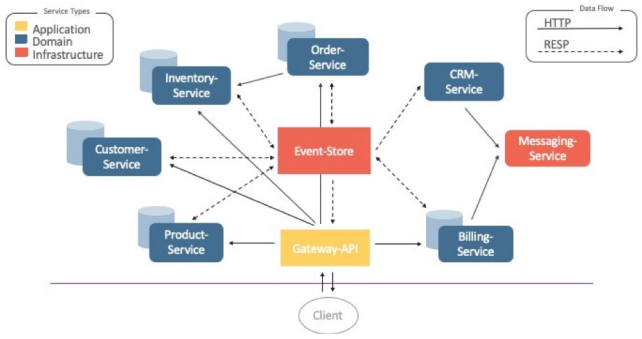

Figure 4: OrderShop architecture

The OrderShop application's data model includes customers, products, and orders.

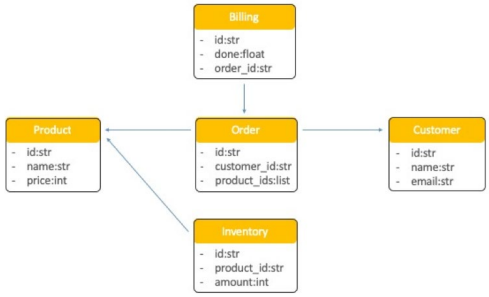

Figure 5: OrderShop data model

HOW TO INSTALL AND RUN ORDERSHOP

1. DClone the repository from here.
2. Make sure you have already installed both Docker Engine and Docker Compose.
3. Install Python3.
4. Start the application with `docker-compose up`.
5. Install the requirements with `pip3 install -r client/requirements.txt`.
6. Then execute the client with `python3 -m unittest client/client.py`.
7. Stop the CRM service with `docker-compose stop crm-service`.
8. Re-execute the client and you'll see that the application functions without any error.

UNDER THE HOOD

Below are some sample test cases in `client.py`, along with corresponding Redis data types and keys.

TEST CASE	DESCRIPTION	TYPES	KEYS

`test_1_create_customers`	Creates 10 random customers	Set
		Stream
		Hash
			`customer_ids`
			`events:customer_created`
			`customer_entity:`*`customer_id`*

`test_2_create_products`	Creates 10 random product names	Set
		Stream
		Hash
			`product_ids`
			`events:product_created`
			`product_entity:`*`product_id`*

`test_3_create_inventory`	Creates inventory of 100 for all products	Set
		Stream
		Hash
			`inventory_ids`
			`events:inventory_created`
			`inventory_entity:`*`inventory_id`*

`test_4_create_orders`	Creates 10 orders for all customers	Set
		Stream
		List
			`order_ids`
			`events:order_created`
			`order_product_ids:<>`

`test_5_update_second_order`	Updates second order	Stream	`events:order_updated`
`test_6_delete_third_order`	Deletes third order	Stream	`events:order_deleted`
`test_7_delete_third_customer`	Deletes third customer	Stream	`events:customer_deleted`

`test_8_perform_billing`	Performs billing of first order	Set
		Stream
		Hash
			`billing_ids`
			`events:billing_created`
			`billing_entity:`

`test_9_get_unbilled_orders`	Gets unbilled orders	Set Hash	`billing_ids`, `order_ids` `billing_entity:`*`billing_id`*, `order_entity:`*`order_id`*

Figure 6: OrderShop test cases and corresponding Redis data types and keys

We chose the Redis Streams data type to save OrderShop's events because the abstract data type behind them is a transaction log, which perfectly fits the continuous event stream use case. We selected different keys to distribute the partitions and chose to generate an entry ID for each stream. This consists of the timestamp in seconds "-" microseconds in order to be unique and preserve the order of events across keys/partitions.

```
127.0.0.1:6379> XINFO STREAM events:order_created
 1) "length"
 2) (integer) 10
 3) "radix-tree-keys"
 4) (integer) 1
 5) "radix-tree-nodes"
 6) (integer) 2
 7) "groups"
 8) (integer) 0
 9) "last-generated-id"
10) "1548699679211-658"
11) "first-entry"
12) 1) "1548699678802-91"
    2) 1) "event_id"
       2) "fdd528d9-d469-42c1-be95-8ce2b2edbd63"
       3) "entity"
       4) "{\"id\": \"b7663295-b973-42dc-b7bf-
8e488e829d10\", \"product_ids\": [\"7380449c-d4ed-
41b8-9b6d-73805b944939\", \"d3c32e76-c175-4037-ade3-
ec6b76c8045d\", \"7380449c-d4ed-41b8-9b6d-73805b944939\",
\"93be6597-19d2-464e-882a-e4920154ba0e\",
\"2093893d-53e9-4d97-bbf8-8a943ba5afde\",
\"7380449c-d4ed-41b8-9b6d-73805b944939\"], \"customer_
```

CODE CONTINUED ON NEXT COLUMN

```
id\": \"63a95f27-42c5-4aa8-9e40-1b59b0626756\"}"
13) "last-entry"
14) 1) "1548699679211-658"
    2) 1) "event_id"
       2) "164f9f4e-bfd7-4aaf-8717-70fc0c7b3647"
       3) "entity"
       4) "{\"id\": \"1ea7f394-e9e9-4b02-8c29-
547f8bcd2dde\", \"product_ids\": [\"2093893d-53e9-4d97-
bbf8-8a943ba5afde\"], \"customer_id\": \"8e8471c7-2f48-
4e45-87ac-3c840cb63e60\"}"
```

We chose Redis Lists and Hashes to model the entity cache as a simple projection of the domain model (product, order, customer):

```
127.0.0.1:6379> TYPE customer_IDs
list

127.0.0.1:6379> LRANGE customer_IDs 0 10
1) "3b1c09fa-2feb-4c73-9e85-06131ec2548f"
2) "47c33e78-5e50-4f0f-8048-dd33efff777e"
3) "8bedc5f3-98f0-4623-8aba-4a477c1dd1d2"
4) "5f12bda4-be4d-48d4-bc42-e9d9d37881ed"
5) "aceb5838-e21b-4cc3-b59c-aefae5389335"
6) "63a95f27-42c5-4aa8-9e40-1b59b0626756"
7) "8e8471c7-2f48-4e45-87ac-3c840cb63e60"
8) "fe897703-826b-49ba-b000-27ba5da20505"
9) "67ded96e-a4b4-404e-ace6-3b8f4dea4038"

127.0.0.1:6379> type customer_entity:67ded96e-a4b4-404e-
ace6-3b8f4dea4038
hash

127.0.0.1:6379> HVALS customer_entity:67ded96e-a4b4-
404e-ace6-3b8f4dea4038
1) "67ded96e-a4b4-404e-ace6-3b8f4dea4038"
2) "Ximnezmdmb"
3) "ximnezmdmb@server.com"
```

If you'd like to dig deeper, you can learn more about Redis Streams here.

Redis Enterprise for Microservices

Redis Enterprise, built over open-source Redis, offers a flexible deployment model — you could deploy Redis Enterprise close to your microservice:

- In your own data center, be it on-premises (VMs or bare metal) or in the cloud.
- In a containerized environment, orchestrated by Kubernetes or other container orchestrators.
- In a cloud-native/PaaS environment like PCF or OpenShift.

Being a multi-tenant solution, Redis Enterprise isolates the data between microservices. Most of all, it allows you to tune your database to maintain a trade-off between performance and data consistency/durability.

Redis Enterprise is also a CRDTs (conflict-free replicated data types)-based, active-active database. Redis CRDTs account for the possibility of multiple instances of a microservice by allowing each microservice to connect to the local instance of a distributed Redis Enterprise database (event store). This CRDT technology ensures strong eventual consistency, which means that all data replicas will eventually achieve the same consistent state across all microservices.

Redis Enterprise also features higher throughput (number of operations per second) and lower latency. Its shared-nothing database architecture can perform a million read/write operations per second with just two commodity cloud instances while making sure the data is also durable. In comparison, other databases in the market require significantly more servers/cloud instances to deliver the same throughput.

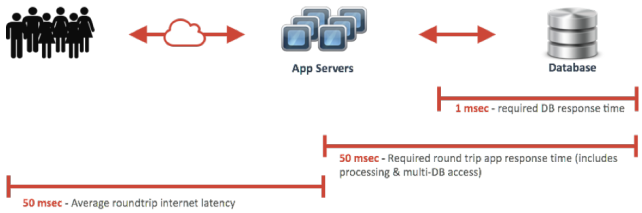

Figure 7: 100 msec — the new standard for E2E application response time, under any load

Conclusion

Redis Streams data structure combines the benefits of pub/sub, lists, and sorted sets. It allows asynchronous data transfer, supports connectors for multiple clients, and stays resilient to connection loss.

With Redis Enterprise, your microservices get a highly reliable event store database with always-on availability and durability options ranging from hourly snapshots to log changes every second (with Append Only File (AOF) every second) or every write (with AOF every write). You can configure Redis Enterprise to have a combination of in-memory replication and persistence. These configurable options help you optimize the performance of your microservices for the type of data they handle — ephemeral, transient, operational, or transactional.

Written by **Shabih Syed,** *Sr. Director of Product Marketing at Redis Labs*

Shabih is a Sr. Director of Product Marketing at Redis Labs. He has over 10 years of experience with product marketing and product management of cloud-based application integration and data management platforms. Most recently, he was head of product marketing at Liaison Technologies and led a successful exit to OpenText. Previously, he held senior management roles at Prysm, HP, and IBM.

Written by **Martin Forstner,** *Solution Architect at Redis Labs*

Martin worked as a software engineer for 11 years before he joined Redis Labs as a solution architect. He has developed server and client-side solutions in several languages and paradigms. Martin specializes in distributed systems programming using appropriate networking protocols and database technologies. In addition, he has experience in architecting and deploying applications in different cloud environments and ensuring their smooth operation.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300