WHITE PAPER

Eight Secrets to Metering with Redis Enterprise

The Complete Guide to Real-time Metering at Scale

Roshan Kumar, Redis

CONTENTS

Abstract

Applications that utilize complex functionality such as consumption based pricing, resource usage limits, measuring/counting at scale and normalizing resource usage (such as traffic shaping) require extremely high performance and low latencies, particularly when dealing with streaming data. Redis Enterprise (https://redis.com/why-redis/redis-enterprise/) simplifies the implementation of metering providing high performance, and scaling, even at very high data volume and velocity. This whitepaper addresses many common challenges you may face when working with metering algorithms and how Redis Enterprise solves these effortlessly and with great resource efficiency.

What Is Metering

Metering is not just a simple counting problem. Metering is often confused with measuring, but is usually more than that. Metering does involve measuring, but as an ongoing process. According to the free dictionary, a meter is defined as:

"Any of various devices designed to measure time, distance, speed, or intensity or indicate and record or regulate the amount or volume, as of the flow of a gas or an electric current."[1]

As the definition suggests, metering also involves regulating something based on what's being measured as an ongoing process. Modern applications incorporate metering in many different ways, ranging from counting people, objects, or events to regulating usage, controlling access and allocating capacity.

Common Metering Scenarios

Metering is required in applications that deal with scenarios such as:

1. **Consumption-based pricing models:** Unlike one-time or subscription-based payment models, consumption-based pricing models allow consumers to pay only for actual usage. Consumers enjoy greater flexibility, freedom, and cost savings while providers gain greater consumer retention.

   Implementing such models can be tricky. Sometimes the metering system has to track many items of usage and many metrics in a single plan. For example, a cloud provider may set different pricing levels for CPU cycles, storage, throughput, number of nodes, time, etc. while a telecommunications company may set different levels of allowed consumption for minutes, data, text etc. The resulting consumption-based pricing may include a variety of terms such as "Pay as you go," "freemium," "tiered pricing," etc. The metering solution must enforce capping, charging or extending services depending on the type of consumption-based pricing.

2. **Restricting resource utilization:** Every service on the internet can be abused through excessive usage unless that service is rate limited. Popular services such as Google AdWords API and Twitter Stream API incorporate rate limits for this reason. Some extreme cases of abuse lead to denial of service (DoS). To prevent abuse, services and solutions that are accessible on the internet must be designed with proper rate limiting rules. Even simple authentication and login pages must limit the number of retries for a given interval of time.

   Another example where restricting resource utilization becomes necessary is when changing business requirements put greater load on legacy systems than they can support. Rate limiting the calls to the legacy systems allows businesses to adapt to growing demand without needing to replace their legacy systems.

   In addition to preventing abuse and reducing load, good rate limiting also helps with the management of bursty traffic scenarios. For example, an API enforcing a brute force rate-limiting method may allow 1000 calls every hour. Without a traffic-shaping policy in place, a client may call the API 1000 times in the first few seconds of every hour. This may not be a desirable situation because it might exceed what the infrastructure can support. Popular rate-limiting algorithms such as Token Bucket and Leaky Bucket prevent bursts by not only limiting the calls, but also distributing them over time.

3. **Resource distribution:** Congestion and delays are common scenarios in applications that deal with packet routing, job management, traffic congestion, crowd control, social media messaging, data gathering, etc. Queueing models offer several options for managing the queue size based on the rate of arrival and departure, but implementing these models at large scale isn't easy.

   Backlog and congestion can get really daunting while dealing with fast data streams. Clever designers need to define acceptable queue length limits, while incorporating both the monitoring of queuing performance and dynamic routing based on queue sizes.

4. **Counting at scale for real-time decision making:** E-commerce sites, gaming applications, social media, mobile apps, etc. attract millions of daily users. For many sites, more eyeballs yield greater revenue. Therefore, counting visitors and their actions accurately is critical to business. Real-time charts and command & control dashboards drive their decision-making processes. Counting is also useful for many other use cases such as error retries, issue escalation, DDoS attack prevention, traffic profiling, on-demand resource allocation, fraud mitigation, etc.

Metering Design Challenges

Architects of metering solutions have to consider many parameters for an ideal solution, including:

5. **Design complexity:** Counting, tracking and regulating volumes of data--especially when they arrive at a high velocity--is a daunting task. Solution architects can handle metering at the application layer by using programming language structures. However, such a design is not resilient to failures or data loss. Traditional disk-based databases are robust, and promise a high degree of data durability during failures. But not only do they fall short of providing the requisite performance, they also increase complexity without the right data structures and tools to implement metering.

6. **Latency:** Metering typically involves numerous, constant updates to counts. Network and disk read/write latency adds up while dealing with large numbers. This could snowball into building up a huge backlog of data leading to more delays. The other source of latency is a program design that loads the metering data from a database to the program's main memory, and writes back to the database when done updating the counter.

7. **Concurrency and consistency:** Architecting a solution to count millions and billions of items can get complex when events are captured in different regions, and they all need to converge in one place. Data consistency becomes an issue if many processes or threads are updating the same count concurrently. Locking techniques avoid consistency problems and deliver transactional level consistency, but slow down the solution.

8. **Durability:** Metering affects revenue numbers, which implies that ephemeral databases are not ideal in terms of durability. An in-memory datastore with durability options, either via replication or persistence, is a perfect choice.

9. **Stability:** Some meters start small, but get sudden traction. They go from tracking about a thousand items per second to millions in just a matter of days. Adding external layers to keep up with the load to attain stability, HA, better performance, etc., increases the number of points for failure.

The cache's main purpose is to reduce the time needed to access data stored outside of the application's main memory space. Additionally, caching is also an extremely powerful tool for scaling up external data sources and mitigating the effects that usage spikes have on them.

An application-side cache effectively reduces all resource demands needed to serve data from external sources, thus freeing these resources for other uses. Without the use of a cache, the application interacts with the data source for every request, whereas when a cache is employed only a single request to the external data source is needed, with subsequent access served from the cache.

Additionally, a cache also contributes to the application's availability. External data sources may experience failures that result in degraded or terminated service. During such outages the cache can still serve data to the application and thus retain its availability.

How Redis Enterprise Solves Common Metering Problems

Redis Enterprise (https://redis.com/why-redis/redis-enterprise/) technology encapsulates open-source Redis, while fully supporting all of its commands, data structures, and modules. It adds exceptional flexibility, stability, high performance, and unmatched resilience with multiple deployment choices.

1. High Performance with Low Latency

Depending on the scale of the solution, counting and metering can involve thousands, if not millions of updates to the database every second. The primary requirements of a database to support such a solution are:

a. Deliver high throughput for write operations
b. Respond with low, sub-millisecond latency
c. Be cost effective--deliver the above two benefits using the least amount of computational resources

Redis Enterprise delivers the highest throughput at the lowest latency. Avalon Consulting benchmarked a comparison of database throughput and latency among the popular NoSQL databases for a high-volume, real-time, write-intensive application. As per the benchmark results, Redis Enterprise surpassed the competition with up to eight times higher throughput and less than half the latency (https://redis.com/docs/nosql-performance-benchmark).

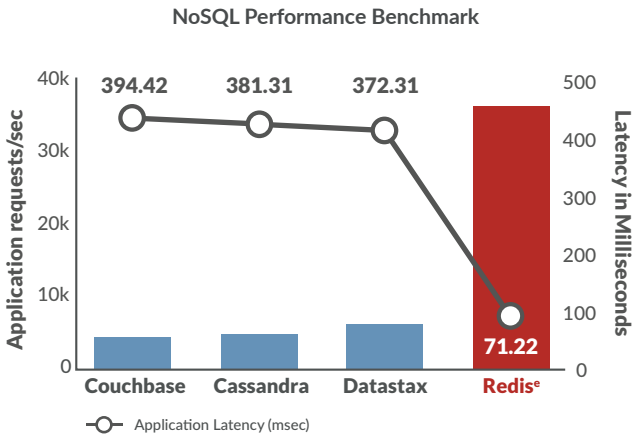

Figure 1. Independent performance benchmark by Avalon Consulting. Redis Enterprise delivers the highest throughput with the lowest latency

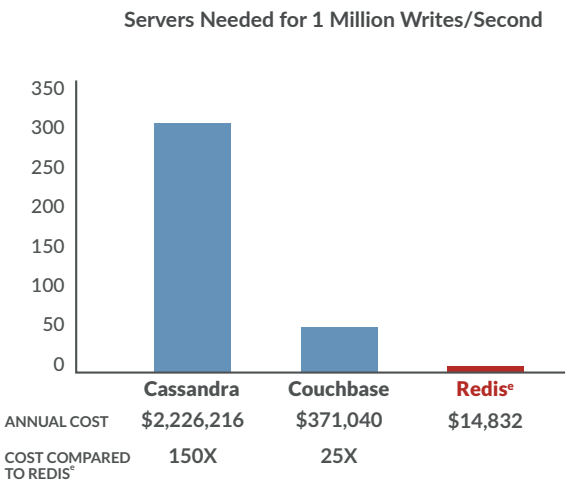

Figure 2. *Redis Enterprise delivers the maximum throughput with the lowest number of servers, slashing operational costs by up to 99% (Google Cloud Platform performance benchmarks: https://cloudplatform.googleblog.com/2015/04/a-guy-walks-into-a-NoSQL-bar-and-asks-how-many-servers-to-get-1Mil-ops-a-second.html)*

2. Atomic Commands for Counting

Redis provides commands to increment values without the requirement of reading them to the application's main memory.

Command	Description
INCR key	Increment the integer value of a key by one
INCRBY key increment	Increment the integer value of a key by the given number
INCRBYFLOAT key increment	Increment the float value of a key by the given amount
DECR key	Decrement the integer value of a key by one
DECRBY key decrement	Decrement the integer value of a key by the given number
HINCRBY key field increment	Increment the integer value of a hash field by the given number
HINCRBYFLOAT key field increment	Increment the float value of a hash field by the given amount

Table 1. *Atomic Redis commands to increment and decrement values*

Redis stores integers as a base-10 64-bit signed integer, therefore the maximum limit for an integer is a very large number: $2^{63} - 1 = 9{,}223{,}372{,}036{,}854{,}775{,}807$.

3. Built-in Time-To-Live (TTL) on the Keys

One of the common use cases in metering is to track usage against time and to limit resources after the time runs out. In Redis, one can set a time-to-live value for the keys. Redis will automatically disable the keys after a set time-out. The following table lists several methods of expiring keys.

Command	Description
EXPIRE key seconds	Set a key's time to live in seconds
EXPIREAT key timestamp	Set the expiration for a key as a UNIX timestamp
PEXPIRE key milliseconds	Set a key's time to live in milliseconds
PEXPIREAT key timestamp	Set the expiration for a key as a UNIX timestamp in milliseconds
SET key value [EX seconds]	Decrement the integer value of a key by the given number

Table 2. Commands to expire keys in Redis

The messages below give you the time-to-live on the keys in terms of seconds and milliseconds.

Command	Description
TTL key	Get the time to live for a key
PTTL key	Get the time to live for a key in milliseconds

Table 3. Redis commands to get the time-to-live

4. Data Structures and Commands for Efficient Counting and More

Redis is loved for its data structures, such as Lists, Sets, Sorted Sets, Hashes, and Hyperloglogs. Many more can be added through its modules API.

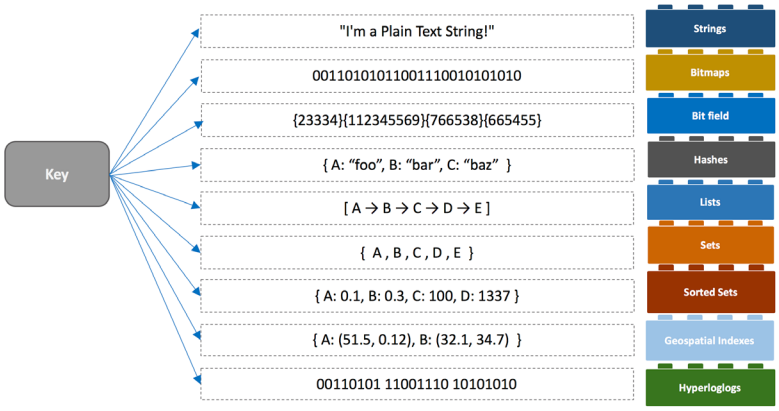

Figure 3. Redis data structures

Redis data structures come with built-in commands that are optimized to execute with maximum efficiency in memory (right where the data is stored). Some data structures help you accomplish much more than the counting of objects. For example, the Set data structure guarantees uniqueness to all the elements. Sorted Set goes one step ahead by not only ensuring that only unique elements are added to it, but also allowing you to order them based on a score. Ordering your elements by time in a Sorted Set data structure, for example, will offer you a time-series database. With the help of Redis commands you could get your elements in a certain order, or delete items that you don't need anymore. Hyperloglog is another special data structure that estimates counts of millions of unique items without needing to store the objects themselves or impact memory

Data Structure	Command	Description
List	**LLEN** key	Get the length of a list
Set	**SCARD** key	Get the number of members in a set (cardinality)
Sorted Set	**ZCARD** key	Get the number of members in a sorted set
Sorted Set	**ZLEXCOUNT** key min max	Count the number of members in a sorted set between a given lexicographical range
Hash	**HLEN** key	Get the number of fields in a hash
Hyperloglog	**PFCOUNT** key	Get the approximate cardinality of the set observed by the Hyperloglog data structure
Bitmap	**BITCOUNT** key [start end]	Counts set bits in a string

Table 4. *Examples of Redis commands for counting using data structures*

5. Data Durability with Persistence and In-Memory Replication

Metering use cases, such as payments, store and update information that is critical to businesses. Loss of data has a direct impact on revenue. It can also destroy billing records, which are often a compliance or governance requirement.

Redis Enterprise allows you to tune consistency and durability based on your data requirements. If you need a permanent proof of record for your metering data, you can achieve durability in more than one way:

1. Persistence – Append only file (AOF - every write, every second), snapshotting

Figure 4a. *Persistence in Redis with AOF on every write*

2. Asynchronous in-memory replication without wait command

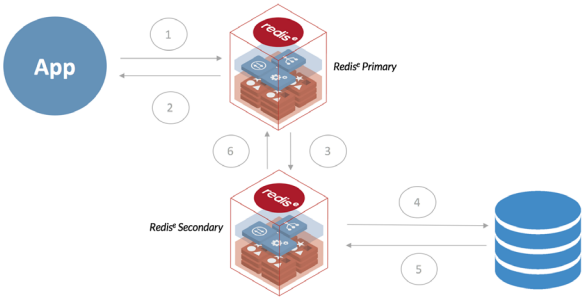

Figure 4b. *Asynchronous replication where the secondary node persists to the disk*

3. Synchronous in-memory replication and persistence with wait command

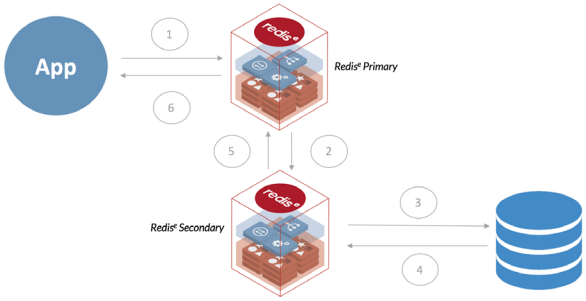

Figure 4c. *Synchronous replication with WAIT command – primary returns to the application only after the secondary node persists the data to the disk*

4. In-memory replication – cross rack, data center, zones and even clouds.

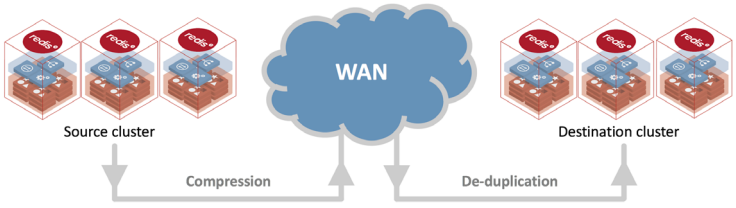

Figure 4d. *In-memory replication across data centers via WAN*

6. High-Availability with Shared-Nothing Dynamic Cluster Architecture

In many metering use cases, the activity of counting and metering never stops. Any failure to record updates—even during a short period of time--may impact the business. Redis Enterprise is built to safeguard stable high performance with full resilience to every type of failure scenario

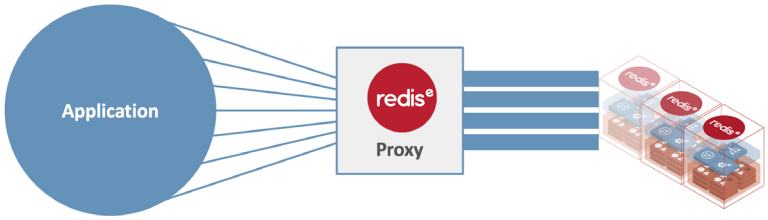

Figure 5. *Redis Enterprise's Shared-Nothing Dynamic Cluster Architecture*

Redis Enterprise's shared-nothing dynamic cluster architecture offers many benefits to your metering solution:

- The architecture delivers fully-automated high availability, instant automatic failure detection, failover without any data loss, backup and recovery
- It utilizes multiple CPU cores for the highest performance with fewest resources
- The zero-latency proxy allows applications to interact with a Redis Enterprise cluster as if it were a single instance of Redis. The proxy's architecture decouples the topology and layout of Redis Enterprise from the application. It accelerates throughput by reducing context-switching overhead, maximizing command pipelining, implementing long-lived socket-based persistent connections with Redis instances, and other optimizations.
- The architecture allows Redis Enterprise to oversee all the operations of scaling, sharding, re-sharding and migration across nodes in a fully automated manner.

7. Simplified Design Due to Built-in Lock-Free Architecture

Redis processing is single threaded; this ensures data integrity, as all the write commands are automatically serialized. If your application runs in multiple threads, then the zero-latency proxy will take care of multiplexing all the requests that are going to a particular shard and pipelines all the commands internally to achieve maximum throughput. This architecture relieves the developers and architects from the burden of synchronizing threads in a multithreaded environment.

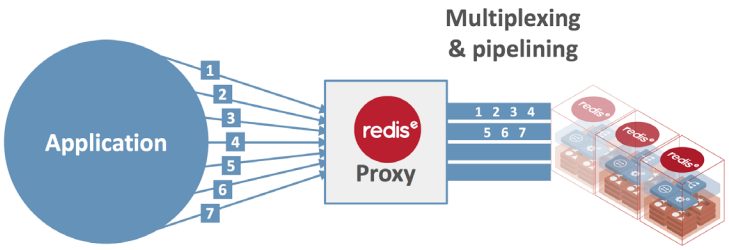

Figure 6. *Multiplexing and pipelining in the zero-latency proxy*

In the case of a popular consumer mobile application, for example, you could expect thousands, and sometimes millions of users accessing the application simultaneously. Let's say the application is metering time used, and two or more users can share minutes concurrently. The parallel threads can update the same object without imposing the additional burden of ensuring data integrity. This reduces the complexity of the application design, and assures speed and efficiency.

8. Support for Multi-master Setup (Coming Soon)

For geographically distributed solutions and distributed architecture involving multiple Redis Enterprise endpoints, Redis Enterprise 5.0 offers a CRDB (Conflict-free Replicated Data Base) with multi-master support [2]. In such a setup, each region will have its own Redis Enterprise master or primary server which connects with other master servers in the topology. Applications connect to their nearest Redis Enterprise cluster to update their counts. Redis Enterprise implements CRDT (Conflict-free Replicated Data Type), ensuring that all master servers exchange data and resolve conflicts, converging to the same state with strong eventual consistency.

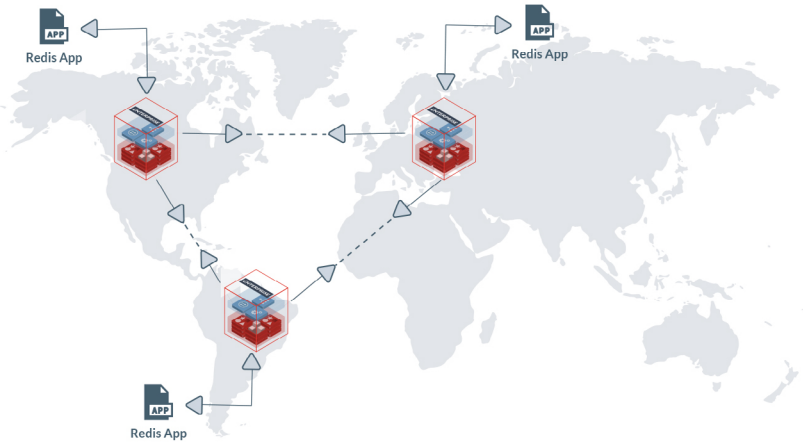

Figure 7. *A sample multi-master topology*

In metering and counting scenarios, applications could connect to a Redis Enterprise cluster deployed in their respective region, and update counts or items from the data structures in their local cluster, while behind the scenes, Redis Enterprise will converge all the updates between all the master servers across all regions.

Sample Implementations

The efficiency of using Redis Enterprise for metering is best demonstrated with some sample code. Several of the scenarios below would require very complex implementations if the database used was not Redis.

1. Blocking Multiple Login Attempts

To prevent unauthorized access to accounts, websites sometimes block users from making multiple login attempts within a stipulated time period. In this example, we restrict the users from making more than three login attempts in an hour using simple key time-to-live functionality.

The key to hold the number of login attempts: `user_login_attempts:<user id>`

Steps:

a. Get the current number of attempts:

```
GET user_login_attempts:<user id>
```

b. If null, then set the key with the expiration time in seconds (1 hour = 3600 seconds):

```
SET user_login_attempts:<user id> 1 3600
```

c. If not null and if the count is greater than 3, then throw an error

d. If not null, and if the count is less than or equal to 3, increment the count:

```
INCR user_login_attempts:<user id>
```

Upon a successful login attempt, the key may be deleted as follows:

```
DEL user_login_attempts:<user id>
```

2. Pay as You Go

The Redis Hash data structure provides easy commands to track usage and billing. In this example, let's assume every customer has their billing data stored in a Hash, as shown below:

```
customer_billing:<user id>
        usage   <actual usage in the unit which the billing is based upon>
        cost    <cost billed to the customer>
        .
        .
```

Suppose each unit costs two cents, and the user consumed 20 units; the commands to update the usage and billing are:

```
hincrby customer:<user id> usage 20
hincrbyfloat customer:<user id> cost .40
```

As you may notice, your application can update the information in the database without requiring it to load the data from the database into its own memory. Additionally, you could modify an individual field of a Hash object without reading the whole object.

Please note: The purpose of this example is to show how to use hincrby and hincrbyfloat commands. In a good design, you avoid storing redundant information such as both usage and cost.

3. Rate Limiting

Rate limiting regulates how frequently a resource is used /accessed by consumers. In the following three examples, we demonstrate increasing levels of sophistication in allocation of resource usage. The examples are all written in Java. In all three cases, cells (the messages or the items) arrive at random times. The business use case sets a limit on how many cells can arrive in a stipulated time window. The algorithm maintains the history of cell arrivals and decides whether the new cell is good to go or not. Figure 6 shows the design of the solution. All the code samples can be accessed at https://github.com/redisdemo/RateLimiter.

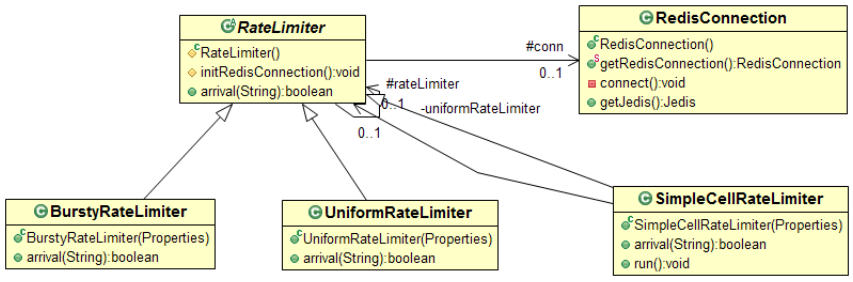

Figure 8. *Class diagram of RateLimiter and its sub-classes*

Example 1: BurstyRateLimiter

In this implementation, the rate limiter allows cells to arrive in bursts. For example, if the use case has a rule that allows 1,000 cells in an hour, this class allows up to 1,000 cells in the first second, itself. In other words, this algorithm doesn't check for bursts, but allows the specified number

```
public class BurstyRateLimiter extends RateLimiter
{
	private String key = "BurstyRateLimiter"; //default key
	private int window = 3600; // in seconds. 1 hr default
	private int actions = 360; // 1 action every 10 seconds
.
.
	// Returns true if the arrival of the cell is within the acceptance
	// rate, false if not.
	public boolean arrival(String cell){

		long currentTime = System.currentTimeMillis();
		long lastWindow = (currentTime - (window * 1000))/1000;

		//delete all messages outside the window
		jedis.zremrangeByScore(key, "0", Long.toString(lastWindow));

		//is zcard less than the allowed number of actions
		long card = jedis.zcard(key);

		//If yes, add message to the sorted set and return true
		if(card < actions){
			jedis.zadd(key, (currentTime/1000), cell);

			return true;
		}

		return false;
	}
}
```

Example 2: UniformRateLimiter

UniformRateLimiter avoids bursts by spreading out the arrival throughout the time window. For example, if the use case allows 60 cells in an hour, this class allows only one cell every minute. This uses the time-to-live feature of Redis keys.

```
public class UniformRateLimiter extends RateLimiter
{

	private String key = "UniformRateLimiter"; //default key
	private int window = 3600; // in seconds. 1 hr default
	private int actions = 360; // 1 action every 10 seconds
	private int interval = window/actions; //
.
.
.
	// Returns true if the arrival of the cell is within the acceptance
	// rate, false if not.
	public boolean arrival(String cell){

		// check if the last message exists.
		long ttl = jedis.ttl(key);
		if(ttl > 0){
			return false;
		}

		// The key lives through the period defined by the interval
		if(key != null && cell != null){
			jedis.setex(key, interval, cell);
			return true;
		}
		return false;
	}
}
```

Example 3: SimpleCellRateLimiter

Unlike the previous two examples, this algorithm queues up all the cells in Redis' List data structure. It then uses the UniformRateLimiter class to determine when to pop a cell from the List.

```
public class SimpleCellRateLimiter extends RateLimiter implements Runnable{
	private String key = "SimpleCell"; //default key

	private RateLimiter uniformRateLimiter = null;

	public SimpleCellRateLimiter(Properties props) throws Exception{
		super();
		props.setProperty("type", RateLimiterFactory.DISTRIBUTION_TYPE_UNIFORM);
		uniformRateLimiter = RateLimiterFactory.getRateLimiter(props);
		Thread t = new Thread(this);
		t.start();
	}

	// This arrival method is different from the previous two examples.
	// In this example, the program queues up the cell in a List
	// data structure. A separate thread checks whether popping the cell
	// is within the acceptance rate.
	public boolean arrival(String cell){
		// Push the cell to the List
		jedis.lpush(key, cell);
		return true;
	}

	public void run(){

		try{
			RedisConnection conn = RedisConnection.getRedisConnection();
			Jedis jedis = conn.getJedis();

			while(true){
				String cell = jedis.rpop(key);
				boolean success = false;
				while(!success){
					success = uniformRateLimiter.arrival(cell);
					if(success){

						// Take action here.....

						System.out.println("Out: "
						 +(System.currentTimeMillis()/1000)+" "
						 +cell);
					}
				}
			}

		}catch(Exception e){
			e.printStackTrace();
		}
	}
}
```

4. redis-cell: Redis Module for Rate Limiting

redis-cell implements the sophisticated Generic Cell Rate Algorithm and provides a single command to manage rate limiting (http://redismodules.com/modules/redis-cell/). It offers a language agnostic rate limiter for applications that have Redis as part of their software stack. As a Redis module, redis-cell enjoys full benefits of Redis' in-memory processing, internal data structures, scalability, and high-availability. It plugs into Open Source Redis 4.0 and Redis Enterprise 5.0 seamlessly.

Conclusion

Redis Enterprise offers powerful tools and capabilities to meet the metering needs of various scenarios. Not only does it provide a stable, resilient architecture crucial to your mission-critical metering solutions, it also offers built-in data structures and commands that simplify counting and metering at scale. The table below summarizes how Redis Enterprise helps you tackle common metering challenges with grace and simplicity.

Metering implementation challenges	How Redis Enterprise addresses the challenge
Design Complexity	Redis Enterprise simplifies the design of a metering solution by offering highly optimized data structures and commands to count and regulate the flow of items at scale. Its zero-latency proxy further enables the applications to scale out using Redis Enterprise's cluster setup as if it were a single Redis instance. This decouples database management complexity from the application design.
Concurrency	Due to its single threaded processing, Redis offers a lock-free architecture by default. Redis Enterprise encapsulates open-source Redis, and enhances this capability in a clustered environment through a server-side, zero-latency proxy that multiplexes and streamlines all the incoming Redis commands.
Consistency	In addition to the lock-free architecture, Redis provides atomic metering commands that ensure data consistency. The multi-master support available in Redis Enterprise 5.0 delivers strong eventual consistency for geographically distributed deployments.
Latency	Redis is benchmarked to deliver millions of ops/sec with sub millisecond latency, with one or two modestly sized server instances. (https://cloudplatform.googleblog.com/2015/04/a-guy-walks-into-a-NoSQL-bar-and-asks-how-many-servers-to-get-1Mil-ops-a-second.html)
Durability	Redis Enterprise delivers durability with in-memory replication and disk based persistence.
Stability	The shared-nothing dynamic cluster architecture delivers high-availability and high performance with full resilience to every type of failure. (https://redis.com/docs/six-essential-features-for-highly-available-redis/)

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redis.com