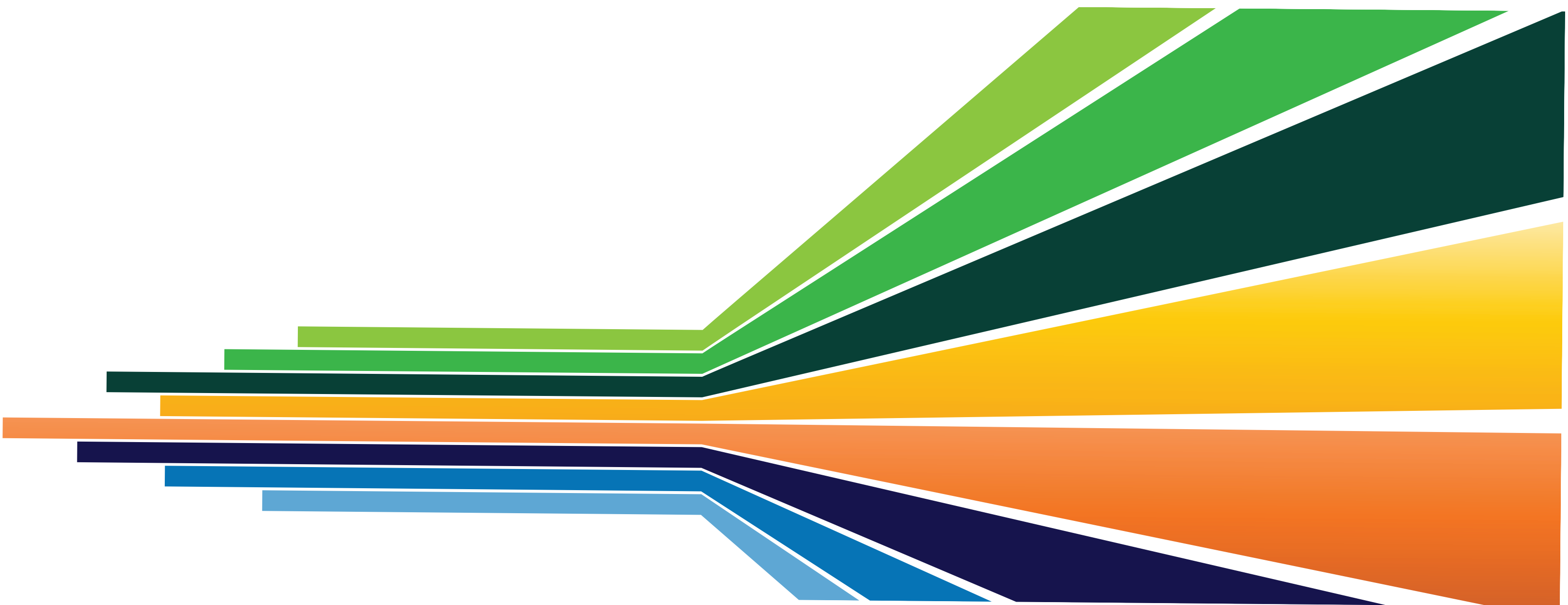
Bartil: Distributed map/list/counter similar to Redis

Abstract: Bartil is a Baratine service that exposes common data structures as REST services. Bartil provides a map, list, tree, string, and counter type that are callable from any client supporting WebSockets or HTTP. You can store any object as the key or value field in the map, list, and tree data types.

Bartil services are persistent; they are stored into Baratine's internal key-value store, io.baratine.core.Store. Saves to the store are batched for high performance and efficiency. Bartil uses a journal to ensure that batched saves are reliable and protected from data loss. Bartil services are addressed by URL. This enables clustering out of the box with no change in code. When Bartil is deployed to a multi-server Baratine pod (virtual cluster), its services become sharded automatically; requests are hashed on the URL and sent to the owning server.

In many ways, Bartil is very similar to Redis; thus, Bartil services should be familiar to Redis users. Bartil shows that you can write any kind of Baratine service that runs about as fast as Redis, but with much more functionality (without having to resort to Lua scripting).



Bartil: a Distributed Data Structure Store on Baratine

Introduction

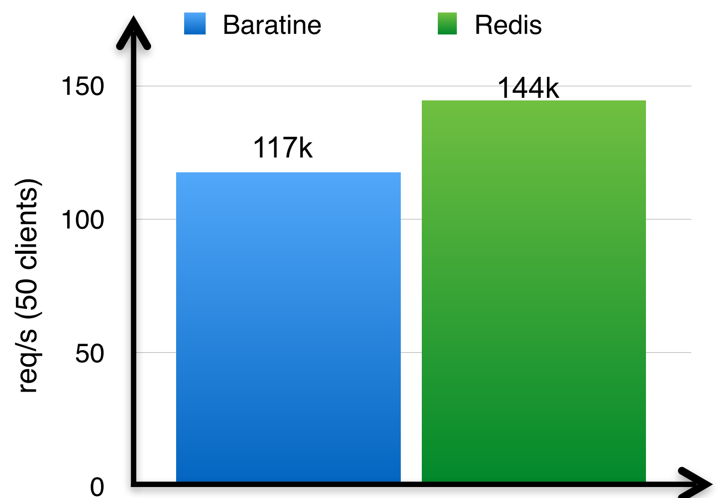
Baratine is a distributed POJO service platform that lets you build object-oriented web services quickly and efficiently. These web services are naturally clustered and can handle millions of operations per second per machine. A wide range of polyglot clients can talk to your web services via WebSockets and HTTP.

To showcase Baratine's POJO-first model, I introduce Bartil, a fast data structure service built on top of Baratine. Bartil is a portmanteau of the words Baratine and *java.util*, emphasizing the rich data structures (such as maps, lists, trees, strings, and counters) that it serves up as web services. Baratine's model provides automatic clustering to Bartil, along with built-in support for both synchronous and asynchronous clients.

What is Bartil?

Bartil provides rich data structures as web services. These data structures are like local libraries and have more functionality than simple key-value stores (i.e. Memcached).

Bartil is like a Java version of Redis, another data structure store written in C, and compares favorably to it in terms of both features and performance. Redis is 23% faster than Bartil. On the other hand, Bartil is easier to customize since it is just Java code. Bartil is like a Java web-app that runs on top of a general-purposed container, which in this case is Baratine.



Understanding Bartil

Baratine services that live at URLs defined by the developer. A client needs a service's URL before it can call the service. For Bartil, its services live at the following base URLs:

- */map*
- */list*
- */tree*
- */string*
- */counter*

Specific instances of a data structure extend off of the base URL. For example, */map/johnsmith1982* and */map/acme-corp* are two different instances. The URLs are how Baratine provides transparent sharding to your services. Baratine hashes the URL to determine which machine to send the request to.

The client interacts with a service instance through a service API. For Java clients, the Java interface is the service API. Here is the */map* interface:

```
public interface MapServiceSync<K,V> extends
MapService {

V get(K key);
List<K> getKeys();

List<V> getValues();
List<V> getMultiple(K ... keys);
Map<K,V> getAll();

boolean containsKey(K key);
boolean containsValue(V value);
int put(K key, V value);
boolean putIfAbsent(K key, V value);

int putMap(Map<K,V> map);
int remove(K key);
int removeMultiple(K ... keys);

boolean rename(K key, K newKey);
int size();
int clear();

boolean delete();
boolean exists();   }
```

Given the above API, a Java client can call the remote service as if it is a plain old Java object:

```
String host = "http://127.0.0.1:8085/s/pod";

ServiceClient client =
ServiceClient.newClient(host).build();

MapServiceSync map = client.lookup("/map/
johnsmith1982")

.as(MapServiceSync.class);

map.put("status", "offline");
```

caucho

The Java client first connects to the Baratine server. Then it creates a proxy for the service at the given URL. The proxy is responsible for serializing Java method calls into messages and sending them over the wire to the Bartil service. Getting the proxy to the service is a two-step process:

1. calling lookup() to create a generic proxy for the service
2. calling as() to create a class-specific API proxy from the generic proxy

Asynchronous methods

The as() method allows a client to cast to whatever API it may choose, as long as the underlying service implementation is compatible. Through this mechanism, different clients can use different APIs - all pointing to the same service.

In addition to the synchronous APIs, Bartil also provides asynchronous APIs for each of its data structures. Generally, the asynchronous methods mirror the synchronous ones:

```
public interface MapService<K,V> {
void get(K key, Result<V> result);
void getKeys(Result<List<K>> result);

void getValues(Result<List<V>> result);
void getMultiple(Result<List<V>> result, K ... keys);
void getAll(Result<Map<K,V>> result);
void containsKey(K key, Result<Boolean> result);
void containsValue(V value, Result<Boolean> result);
void put(K key, V value, Result<Integer> result);
void putIfAbsent(K key, V value, Result<Boolean> result);

void putMap(Map<K,V> map, Result<Integer> result);
void remove(K key, Result<Integer> result);
void removeMultiple(Result<Integer> result, K ... keys);

void rename(K key, K newKey, Result<Boolean> result);
void size(Result<Integer> result);
void clear(Result<Integer> result);

void delete(Result<Boolean> result);
void exists(Result<Boolean> result);
}
```

Baratine maps the synchronous and asynchronous methods to the same implementation, provided the methods are compatible (e.g. same name, same arguments excluding *Result*).

The differences between synchronous and asynchronous methods are that asynchronous methods:

3. return void
4. accept an *io.baratine.core.Result* argument

Asynchronous methods must return void to inform callers that they don’t have to wait for the return value. The *Result* argument is a continuation (a type of

callback) that is to be executed by the caller when the service returns the response. The service may return a response as soon as it receives the request, or anytime in the future.

To use the asynchronous API, you just need to cast the proxy to it in the *as()* call:

```
MapService map = client.lookup("/map/
johnsmith1982").as(MapService.class);

map.put("status", "offline", size ->
{ System.out.println("size is " + size));

});
```

When the return value comes back to the caller, the caller executes the continuation with that value. In the example above, the *Result* is a JDK8 lambda that simply prints out the size of the map.

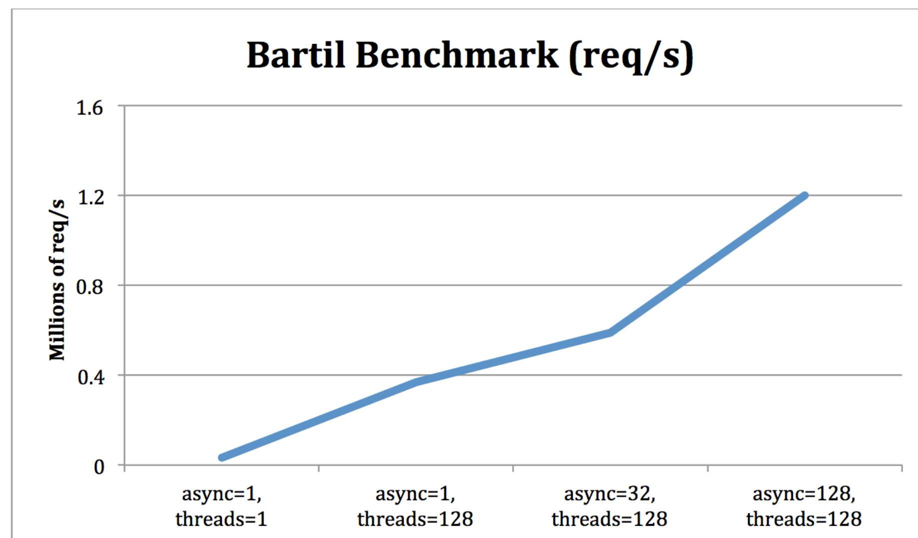
To summarize, the following are powerful features that are unique to Baratine:

1. programming to service APIs
2. using either synchronous or asynchronous APIs

Performance

Bartil can process millions of requests per second per machine because it operates mostly in memory. Baratine allows Bartil to batch multiple writes to the backing store into a large singular write to be written later.

This reduces load dramatically and allows the common path to be in memory and free of writes. The chart below shows that performance increases linearly (a highly desired characteristic) as load increases:



Batching is possible in the first place because of an Inbox that sits in front of every Baratine service. Incoming requests are queued onto the Inbox and the service processes requests from the

queue. This guarantees order and consistency, which makes batching possible, but more importantly, safe.

Summary

Bartil is high-performance data structure service that runs on Baratine, a POJO service platform. It shows the general-purpose nature of Baratine and the wide variety of applications that you can develop with Baratine. Baratine is open source and suitable for cloud applications that require extreme performance.