Polymerase: A Replication Library

Joshua Ma

May 14, 2013

1 Introduction

Polymerase is a Java library that allows developers to replicate Java objects across a set of servers. It presents a simple API to create a replicated version of an object, which is able to transparently repeat method calls across all servers. It also provides facilities to synchronize server initialization and to look up replicated objects that originated from remote servers.

Polymerase guarantees a consistent method calling history for its replicated objects, using an implementation of the Paxos algorithm to achieve fault-tolerant consensus. Additional enhancements allow for relaxed consistency semantics, allowing developers to improve performance at the expense of consistency.

2 Motivation

Modern large-scale systems emphasize speed, reliability, and robustness in order to effectively run the services that depend on them. Distributed systems, in contrast to centralized systems, help achieve these goals through removing certain modes of failure and parallelizing work over multiple machines.

Replication, the synchronization of data across machines, can be an important component in building distributed systems. It improves fault-tolerance, as a given machine can fail as long as its data is replicated to other machines, and it increases accessibility, as more than one machine can serve requests for a given set of data.

A major problem with replication, however, is that it is hard to implement. The few systems that exist for replication or coordination like Apache ZooKeeper require learning and integrating a complex API. Many notable distributed systems, including Chubby, Spanner, and Zookeeper, use Paxos [4] as

an underlying consensus algorithm to synchronize operations. Paxos is difficult to implement, however, and literature that incorporates Paxos often notes this aspect [2, 5].

Polymerase addresses this issue by providing an alternative way of adding replication, through a Java library. The result is a solution that provides less features than a full-fledged system like Zookeeper but makes it much easier to get started with basic replication in a transparent manner.

3 Library Usage

To illustrate usage of Polymerase, we show sample Java code for a replicated messaging service. In our example service, user messages are stored and retrieved through an `Inbox` interface. (We have a concrete Java class `InboxImpl` to implement the interface - it simply keeps an array of messages and appends to / returns the messages as appropriate.)

```
interface Inbox {
    String getOwner();
    List<String> getMessages();
    void append(String message);
}
```

The service consists of a set of Inboxes, one for each user, and provides clients with the ability to add messages and subsequently retrieve them. We'd like to replicate this service across a set of servers: each server presents an identical interface to clients wishing to send/retrieve messages, and data stored at one server is replicated to the others so that clients see consistent messages regardless of which servers they contact. To achieve replication, the service would use Polymerase to replicate the Inbox object for each user. The resulting system is more fault tolerant, since clients can talk to the remaining servers if a given one fails.

We outline the core service below, showing the key procedures that would run with one instance at each server.

```
Replicator replicator;
Map<String,Inbox> inboxes;
```

```
void initialize(int me) {
    replicator = new ReplicatorImpl(servers, me);
    replicator.initialize();
    replicator.setReplicationListener(new ReplicationListener() {
        public void objectReplicated(Object replicatedObject) {
            Inbox inbox = (Inbox) replicatedObject;
            inboxes.put(inbox.getOwner(), inbox);
        }
    });
}

void sendMessage(String sender, String receiver, String message) {
    Inbox inbox;
    if (inboxes.contains(receiver)) {
        inbox = inboxes.get(receiver);
    } else {
        inbox = replicator.replicate(new InboxImpl());
    }
    inbox.append(sender, message);
}

List<String> getMessages(String receiver) {
    Inbox inbox;
    if (inboxes.contains(receiver)) {
        inbox = inboxes.get(receiver);
    } else {
        inbox = replicator.replicate(new InboxImpl());
    }
    return inbox.getMessages();
}
```

In the initialization phase, a new replicator is created by passing in a list of servers in the form of `hostname:port` or `port` if local. A single replicator can be used to later replicate many different types of objects, and each service usually only needs one instance. Calling `initialize()` blocks until all servers have called initialize - this provides a built-in way for developers to synchronize server initialization. The parameter `me` specifies which server the current code is running on - each of n servers would need to be launched with a different integer value from $[0, n)$, specified perhaps via command line arguments. Finally, we set a replication listener that is called whenever an object is replicated (including on a remote server) and stores it in the `inboxes` structure.

Sending a message is simple. If the recipient's inbox does not yet exist, we instantiate a new Inbox and pass it to the `replicate` function. The func-

tion replicates the new instance across all servers and returns an object with the same interface. Calling `inbox.append(sender, message)` results in the `append` method being called on not only the local Inbox but also on the remote, replicated Inboxes. This replication is transparent to the developer - behind the scenes, a proxy intercepts the method call and forwards the replicated call to each replicated instance.

Getting messages is similar, and it should be noted that the replicated method calls can also return values. In this case, `inbox.getMessages()` is replicated despite being free of side-effects. Having the method call enter the consistent history of its peers ensures that it sees the latest appended messages.

Ultimately, incorporating the API consists of adding an initialization stage, setting up a replication listener, and calling `replicate` on newly created objects. After the initial overhead, the replicated version of the object is used identically to the original one. (Due to Java's lack of spport for multiple return values, Polymerase actually returns a value that contains 1) a reference to the replicated instance and 2) its corresponding ID.)

In contrast, performing replication without Polymerase would require essentially re-implementing most of the library, including the Paxos algorithm and a mechanism to access newly replicated objects. Library implementation is described in the next section.

4 Implementation

Polymerase is implemented in pure Java with no dependencies except the Google Guava library for utility functions and concurrent maps. Its main functionality is exposed through the `ReplicatorImpl` class.

4.1 Startup Barrier

On initialize, Polymerase incorporates a barrier to help developers ensure that all servers are alive at the same time. The barrier is intended to block all servers until all servers are at the same initialization phase.

The barrier is implemented through using a thread to send liveliness notifications and a second thread to receive them. Upon starting up, each server sends $n - 1$ notifications to its peers, some of which may not have started up yet. Any peer that is not accepting incoming notifications at the time is skipped

over. When a peer successfully sends a notification to another peer and receives an acknowledgement in response, both mark each other as being alive.

To show correctness, we note that each server is either not accepting responses (while starting up) or accepting responses. After it switches to an accepting response state, it will not change state until all peers are heard from, and servers only start sending notifications after they start accepting them. Given some ordering of servers starting up, for any pair of servers one will always start sending notifications after or at the same time as the other, and as a consequence one will always send after the other has started accepting.

Thus, all pairs of servers will make contact with each other and mark each other as alive. Finally, since no server proceeds until it hears from all of its peers, we have that every server passes the barrier if and only if all servers reach the barrier.

4.2 Object Replication

The initial object is replicated by passing the instance to the `replicate` method on the `ReplicatorImpl` class. Internally, each replicator uses RPC over Java's Remote Method Invocation (RMI) protocol to call methods on its remote counterparts.

To replicate, the replicator generates and assigns a unique ID to the object, passing both ID and object to remote replicators to store. The passing is done over a remote `register` method call - since this is done over RMI, the instance to be replicated is serialized and passed by value instead of by reference. The underlying object is stored in a map at each server and is used for later replicated method calls. The unique ID is used internally to identify the replicated object when it makes calls on remote servers.

4.3 Proxy Handler

After passing the replicated object to each server, the original replicator creates and returns a Java proxy object. The proxy is a Java-specific concept and it implements the same interface as the original replicated object. When methods are called on the proxy, it dispatches them to a replication handler with information about the method called and the passed in arguments. The handler performs the actual replication - it records the method call as an (object, method, arguments) tuple and passes them to all servers. It ensures that all

replicas call the same methods in the same order (see section 4.4 on Paxos) and returns the result of the method call to the original caller.

The proxy matches the interface of the original object, allowing Polymerase to present a transparent replication API. The proxy can be used almost identically to the original (subject to different replication semantics).

4.4 Paxos Replicated State Machines

As a consensus protocol, Paxos operates among a set of peers - in Polymerase the servers are fixed from the start. Each server can operate as a *proposer* that, for a given Paxos instance, attempts to have all nodes, acting as *acceptors*, agree on a generalized "value." Paxos ensures that at most one value can be agreed upon for a given Paxos instance, even in the face of scenarios where less than a majority of servers have failed and where multiple proposers act at the same time. The actual implementation is based on the basic version described in [4]. By identifying each Paxos instance with a unique sequence number, we can have servers agree on a generalized sequence of values corresponding to different sequence numbers.

Polymerase uses Paxos to implemented a replicated state machine (based on the method in [4]) for each replicated object. It is important that all objects have method calls performed in the same order - it is easily seen that two PUTs to the same key in a map, for example, will result in different values depending on the order. Each object starts with a given state, and methods calls are assumed to be the only way to transition between states. (See section 6.2 for a discussion on method call limitations.)

As per the replicated state machine approach, each server maintains a global log of all replicated method calls. (Further optimizations to keep separate logs are discussed in 5.2.) Each log entry is an *Event*, defined by the tuple (object ID, method, arguments), and each Event corresponds to a unique, increasing sequence number corresponding to its position in the log. Each sequence number refers to a given Paxos instance agreeing on a given Event value, which ensures that all peers agree on the same log.

When the proxy handler receives a method call, it translates it into an Event and attempts to log it at the first unused sequence number. If another peer successfully logs the Event at that sequence number first, we try to log at increasing sequence numbers until successful. Storing a unique ID with each Event allows us to verify that a given Event was logged successfully.

Only after logging the Event does the proxy play the logs forward. The practice of log-then-play is necessary in case remote peers have logged Events that aren't currently reflected at the local replica. As an optimization, a background thread periodically plays the logs forward so that a given method call only results in having to play through Events from a bounded period of time. The logs are played forward up to exactly the Event that was logged in the current method call (and no further) - the return value of the last Event is then returned by the proxy in response to the original method call.

To prevent logs from growing indefinitely large, peers periodically communicate to each other the highest sequence number they've played. Once a server hears that all of its peers have processed a given sequence number, it'll know that it can safely discard all log entries up to that sequence number.

4.5 Replication Listener and Object IDs

The replication listener is an interface that the developer implements - it has a single `objectReplicated(Object replicatedObject)` method. The listener is passed to a replicator, and when any object (remote or local) is replicated the replicator calls the `objectReplicated` method with the new replicated instance.

This listener is an important mechanism, as it allows remote peers to easily obtain references to replicated objects. When the listener is called, for example, applications can save a reference to the replicated object to a map, or it may want to start a CPU-intensive process that saves its results to the replicated instance.

Alternatively, Polymerase also returns the IDs of replicated objects, and programs can use `replicator.lookup(id)` to fetch a replicated object. However, developers are left with the problem of communicating the object ID to other servers, which can be done through non-replicated networking calls.

5 Consistency Model

By default, Polymerase uses a sequential consistency model applied across all replicated objects. Replicated method calls are logged in the same global log, so if `A.foo()` is called before `B.bar()` on the same server then it will hold true in the replicated result. Depending on network conditions, though, calling two methods on separate servers does not guarantee any ordering - thus we only provide sequential consistency. There are many times that global sequential

consistency isn't necessary, though, and we provide additional ways of relaxing consistency.

5.1 @AllowStale

Inspired by the consistency model in PNUTS [3], Polymerase allows developers to indicate that certain methods can be called on *stale* copies of replicated objects. Under normal operation, Polymerase will ensure that all preceding operations will have been applied to the given object before calling a repliced method. If the developer uses the Java annotation of `@AllowStale` on a method, however, Polymerase calls the method immediately without waiting for the full log to be played.

There is therefore some complexity foisted upon the developer - methods should only be annotated as `@AllowStale` if the method does not mutate object state or if the ordering doesn't affect the final replicated state. Otherwise, calling a method out of order could result in objects no longer being fully replicated. In the `RelaxedHashMap` below, the `get` method is marked with `@AllowStale`. Write methods like `put` still follow sequential consistency, but read-only methods like `get` are now best effort. Performance improves because the call can be completed without waiting for all other replicated method calls to be played through first. This may be useful for a cache, for example, where it's okay for a `get` re-ordered before a `put` to cause a cache-miss.

```
class RelaxedHashMap {
    Map<String,String> backingMap = new HashMap<String,String>();

    @AllowStale
    String get(String key) {
        return backingMap.get(key);
    }

    void put(String key, String value) {
        backingMap.put(key, value);
    }
}
```

Stale method calls can be especially useful because, at a given server, they are guaranteed to see a state equal to or after the state observed by a previous stale method call. (The log can only play forward.) Take, for example, the

social media example similar to the one posited in the PNUTS paper [3]. We have a service where users share text updates to the network and simultaneously manage privacy settings, maintaing a set of users who are allowed to view these messages. We want to handle the case where a user A can remove viewing permissions from a second user B, post a secret update S, and is guaranteed that B cannot read S.

Using Polymerase, we might have a replicated object that represents a given user's state on the service - a simplified version would consist of a set of users (to capture privacy settings) and a list of strings to capture the user's updates.

```
class ServiceUser {
    List<String> updates;
    Set<ServiceUser> trustedUsers;

    void givePermissions(User user) {
        privacySettings.add(user);
    }

    void removePermissions(User user) {
        privacySettings.remove(user);
    }

    void addUpdate(String update) {
        updates.add(update);
    }

    @AllowStale
    boolean allows(User viewer) {
        return privacySettings.contains(viewer);
    }

    @AllowStale
    List<String> getUpdates() {
        return updates;
    }
}
```

The method `givePermissions` and `removePermissions` add and remove the specified user to the set of users with permissions. We assume a higher layer of software is checking permissions, and that it calls `allows` first to check if a given viewer is allowed to see the object's updates. We assume that, if and only if `allows` returns true does the code then call `getUpdates` to return all updates

(regardless of permissions). The latter two methods are marked `@AllowStale` without issue.

In the case of the above example, we would have called `userA.removePermissions(userB)` followed by `userA.addUpdate(secret)`.

Our retrieval code would look like

```
List<String> updates = userA.getUpdates();
if (userA.allows(userB)) {
    return updates;
}
return new ArrayList<String>();
```

We want to ensure that this code returns either 1) a list of updates prior to *A* posting `secret`, or 2) nothing at all. If the updates do indeed contain `secret`, then the call to `allows` is guaranteed to see a state *after* the secret update was added, with permissions removed. The call will return false, and no updates will be returned. If the updates don't contain the secret update, the call to `allows` may or may not see the permission change. Regardless, though, either no updates will be returned or the updates without the secret will be returned. The code is correct with respect to the desired privacy restrictions.

It's important to note that reversing the order of `allows` and `getUpdates` would *not* result in correct behavior, as it's possible for the secret update to be applied in between the two calls.

To implement this, twhen calling a method the proxy handler first checks if the `@AllowStale` annotation is present. If present, the handler invokes the method directly - if not, it logs the Event and plays the log as described in 4.4.

5.2 Parallelization

Polymerase maintains global consistency, meaning the ordering of method calls on *all replicated objects* is consistent between servers. All replicated calls must therefore be serialized, which may not be necessary or desired in all cases. Polymerase allows for two degrees of parallelization, specified by an annotation on *a given Java class*. `@ClassParallel` enforces serialization between only method calls of that class. For example, putting `@ClassParallel` on a `BankAccount` class would ensure all deposits and withdrawals are serialized consistently, across different accounts.

A second annotation, `@InstanceParallel`, enforces serialization at the instance level. The annotation would be useful for map-like objects, for example, where you want GET and PUT requests to be serialized on a given map without impacting other replicated maps.

Parallelization is implemented by using separate logs, with one log for each serialized group. Each log has its own set of sequence numbers - for a given method invocation, Polymerase look at the annotation on the corresponding class to determine which Paxos-backed log to record it in. Paxos proposals are then identified by (log ID, sequence number) instead of just sequence number, and a different value is allowed to be agreed upon for each unique combination of log and sequence number.

6 Replication Semantics

While Polymerase attempts to make the replication process as transparent as possible in terms of the developer-facing API, there are notable semantic differences that come with replication.

6.1 Timing

Method calls on replicated instances are not invoked immediately, since they're placed in a log. The background thread that periodically plays the log forward helps to ensure some recency, but any slow method call can indefinitely delay the remaining calls. Polymerase guarantees a consistent ordering of method calls, but the actual timing can vary heavily between replicas.

6.2 Assumptions on Method Calls

Polymerase assumes that, given the replicated object as a state machine, calling the same methods with the same parameters (copied by value) will produce the same state transitions on replicas. This requires that method calls do not depend on the system time that they're called at, for example, and all other object state can only be changed via replicated calls.

In practice, most well-encapsulated classes will not run into replication issues. With the exception of I/O calls to save to external filesystems or databases (see section 6.4), if replicated objects 1) do not rely on global state and 2) do not leak internal object references then objects will be correctly replicated.

6.3 Serialization Requirements

Because of the underlying mechanisms used, objects involved in replication must be serializable. Objects to be replicated must be sent to remote peers, and parameters, passed in method calls on replicated objects, are sent to remote peers as well. However limiting this might seem, using immutable objects for fields and parameters is an easy way to avoid these serialization issues.

Additionally, the API requires that replicated concrete classes implement a corresponding Java interface, as the underlying Java proxy can only be cast to an interface and not to a concrete class. It's a common and encouraged Java practice to refer to objects by their interface [1] for better testability and flexibility in general, so this does not present a major issue. Any existing classes that can't be modified to have an interface (by inclusion through a library, for example) can always be composed into a new class that delegates to the old and implements a corresponding interface.

Replicated method calls *can* reliably call other methods on non-replicated objects (that were passed in as parameters). Even though the parameter objects' methods aren't replicated, the replicated calls will call them at each peer.

However, replicated objects currently cannot be passed into other replicated method calls, which require parameters to be serializable. Replicated objects are not serializable because they keep references to the replicator at their current peer. It should be possible to override the default serialization, though, by "serializing" the replicated object as just its object ID. When deserialized at a given peer, the deserialization procedure can read the object ID, find the local instance of the object, and return a properly proxied object similar to normal replication. However, calling a replicated call `foo()` from a parent replicated call `bar()` leads to its own issues. It's not common for a developer with n replicas to intend for n^2 replicated calls to `foo()` to occur, since each of n cals to `bar()` would spawn another n calls.

6.4 Interaction with Non-Replicated Resources

Unfortunately, any replicated method calls that interact with non-replicated resources like databases or files can be problematic if non-replicated objects are reading/writing from the same files. Due to the timing differences described above, the time at which other processes read/write to files can result in inconsistent behavior at different replicas.

Additionally, replica calls that interact with a shared resource (like a single

database) will result in n calls instead of 1. Developers could work around this, though, by introducing a common intermediary service that all replicated methods call. By passing a unique ID to a replicated call that gets forwarded to the intermediary service, we can identify and drop duplicate method calls.

7 Fault Recovery

Although it can continue to operate as long as a majority of peers have not failed (thanks to Paxos), Polymerase does not yet implement a fault recovery mechanism. Peers could recover from failure by re-creating replicated objects during its initialization phase, where the new peer fetches copies of the replicated objects from an existing peer. Once the new peer learns the last sequence number that has been processed corresponding to the newly fetched objects, it can resume operation. If the peer it had contacted happened to be behind and transferred stale objects, the new peer will quickly learn of new log entries when it tries to log events of its own through Paxos. Even if it doesn't log its own events, after the new peer takes part in other Paxos instances it will create gaps in its logs corresponding to the missing log entries. When the background thread to play forward entries sees missing sequence numbers, it will query peers to discover the missing values. (An easy way to query the peers is to try to log a NO-OP event and see the actual event that peers return.)

8 Conclusion

Polymerase provides a simple library to enable replication in Java server applications. It provides transparent replication through using Java proxies and conforms to basic, overly strict consistency model by default. Through the use of annotations, developers can indicate places where weaker consistency is desired for the sake of better performance. Additionally, Polymerase provides barrier and replication listener mechanisms to make it simple to synchronize server initialization and to acquire references to replicated objects, respectively.

The transparency of Polymerase does make it more difficult to reason about more complicated objects when replicated. Well-encapsulated objects that expose immutable datatypes and do not depend on global state are "replication safe," but replication is more difficult when replicated instances depend on non-replicated resources. Although this limits the places where Polymerase can be

used, it still remains a compelling option to automatically replicating simple data across a system.

References

[1] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[2] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live-an engineering perspective (2006 invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing-PODC*, volume 7, 2007.

[3] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[4] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[5] Robbert Van Renesse. Paxos made moderately complex, 2011.