# JPaxos – User Guide

*Release 1.0*

Jan Kończak, Tomasz Żurkowski,
Nuno Santos, Paweł T. Wojciechowski
www.it-soa.pl/jpaxos

January 2011

# Contents

**Contents:**

CHAPTER 1

# Overview

JPaxos is a Java library and runtime system for efficient state machine replication. With JPaxos it is very easy to make a user-provided service tolerant to machine crashes. Our system supports the *crash-recovery* model of failure and tolerates message loss and communication delays.

**State machine replication** is a general method for implementing a fault-tolerant service by replicating it on separate machines and coordinating client interactions with these replicas (or copies). The physical isolation of machines in a distributed system ensures that failures of server replicas are independent, as required. As long as there are enough of non-faulty replicas, the service is guaranteed to be provided.

JPaxos makes the following assumptions about the replicated service:

- deterministic behaviour, i.e. multiple copies of the service begun in the start state, receiving the same inputs in the same order will arrive at the same state having generated the same outputs
- non-Byzantine failures, i.e. a service machine can only crash
- crash-recovery supported, i.e. after crash, the service can be restarted with the same IP address.

## 1.1 Two execution modes

JPaxos can be run in one of the following two modes of operation (the mode is set up from the configuration file at system start up and cannot be changed at runtime):

Basic mode

- to be able to tolerate $f$ faulty replicas, the system must consists of at least $2f + 1$ replicas (i.e. $n \geq 2f + 1$)
- after replica crash, it recovers its state from other replicas

Extended mode

- no limit on the number of faulty processes (i.e. $f = n$)
- after replica crash, it recovers its state from non-volatile memory
- around 200 times slower than the basic mode

## 1.2 Simple API

For the end user JPaxos provides:

1. `Service` interface – it specifies a few methods for interfacing the user-defined service code with JPaxos; our library provides abstract classes implemneting `Service` interface, making it easier to to create a new service (state machine); see *Service interface* for details
2. `Replica` class – that should be instantiated and started (see *Replica* for details)
3. `Client` class – the part that may send request to be executed (see *Client* for details)

The `Service` and `Replica` are bound to each other, while `Client` can be located anywhere.

JPaxos guarantees that in crash-recovery model, with static groups, lossy network with any delays:

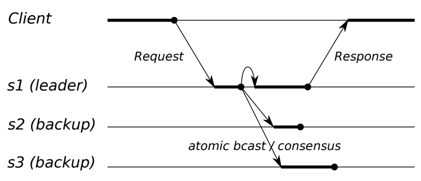
- if the client sends a request, it'll eventually get the answer
- every replica will execute the request exactly once
- any two replicas will execute the requests in the same order

## 1.3 Distributed implementation

JPaxos is a fully distributed implementation, which means that there is no predefined central coordinator that might be a bottle-neck or a single point of failure in the system.

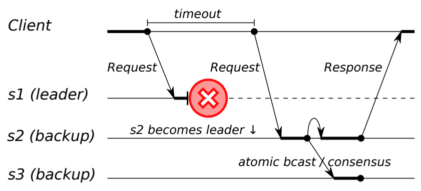
For this, JPaxos implements the Paxos distributed algorithm with various optimizations, in order to efficiently deliver client requests to all service replicas despite any failures. Replicas receive all requests globally ordered, thanks to the total-order (or atomic) broadcast protocol implemented on top of Paxos.

Figure below shows processing of a single request message submitted by the Client to the replicated state machine:



where *Client* is an instance of the `Client` class, while *s1*, *s2*, and *s3* are instances of the `Replica` class.

If a request is delayed or lost due to the network or replica failure, after a timeout, it will be reissued by the *Client*, as illustrated in Figure below:



CHAPTER 2

# Deployment and configuration

This section presents quick overview on what is necessary to compile the library and how to configure the system and start the example which is included in the distribution.

## 2.1 Deployment requirements

JPaxos requires the following components to be installed on the target system:

- Java JRE 1.5 or later

Additionally, the following are optional but helpful tools in compilation and packaging:

- Apache Ant – used to run all build scripts (distributed under the Apache License 2.0)

## 2.2 Configuration file

The format of configuration file used by `Configuration` class is the same as default Java Properties file. As mentioned earlier, this file contains nodes configuration and replica related options.

### 2.2.1 Node configuration

A node is configured with a single line containing `hostname`, `replica port` and `client port`, separated with commas:

```
process.<id> = <hostname>:<replica_port>:<client_port>

process.0 = localhost:2000:3000
process.1 = localhost:2001:3001
process.2 = localhost:2002:3002
```

Above configuration creates three replicas with ids: 0, 1, 2. Replica with id 0, is running on localhost and is using port 2000 to communicate with other replicas and is using port 3000 to accept connections from clients.

### 2.2.2 Crash model selection

One should select the crash model of the system. If the crash model uses the non-volatile memory (i.e. hard drive), the location of the logs may be specified as well.

Currently supported crash models include:

| Type | Name | Needs stable storage | Fault tolerance |
|---|---|---|---|
| CrashStop | CrashStop | No | minority |
| CrashRecovery | FullStableStorage | Yes, heavy usage | catastrophic |
| CrashRecovery | ViewSS | Yes, periodically | minority |
| CrashRecovery | EpochSS | Yes, one write by start | minority |

**Crash model types:**

- CrashStop - once the replica crashed, it cannot recover
- CrashRecovery - the replica may crash and subsequently recover

Fault tolerance ranks:

- minority - the minority of replicas may crash; $f = \lfloor (n-1)/2 \rfloor$
- catastrophic - all replicas may crash; $f = n$

To select a crash model, one must add to the configuration file a line `CrashModel = [crash model name]`. If no crash model is provided, the `FullStableStorage` is assumed.

For choosing a log path one needs to add another line with syntax `LogPath = [path]`. The logs will be actually stored in subdirectory named after replica id in the given location. The default value for log path is `jpaxosLogs`.

An example configuration:

```
CrashModel = ViewSS
LogPath = /mnt/shared/jpaxos/logs
```

### 2.2.3 Replica options

Configuration file also allows to set additional options for replicas. Understanding how each option described below can affect JPaxos is important to achieve high performance.

### 2.2.4 Window size

The window size determines the maximum number of concurrently proposed instances. The meaning of this option is very similar to window size in TCP protocol.

To illustrate it, assume that window size is set to 10. It allows to run instances with id's from 1 to 10 concurrently and to decide instances 2 - 10 before instance 1. JPaxos cannot execute any instance until all previous instances are executed so because instance 1 is not decided / executed, no instance can be executed on state machine. When instances 1 will be decided and executed all consecutive instances will also be executed.

The example above shows that by increasing the value of window size we can decrease the response time - a lot of instances will be decided, but none can be executed. Because of that it is recommended to set this option to lower value and BatchSize to higher value so that decided instances can be executed faster.

**The default value of this option is 2 and can be set using::** WindowSize = 4

### 2.2.5 Batch size

JPaxos will try to batch requests into a single proposal to improve the performance. This option controls the maximum size (in bytes) of requests grouped into one consensus instance.

For example, if maximum batch size is set to 1000 and JPaxos received requests of size 100, 300, 400, 300 bytes, then first three requests will be batched into one consensus instance of size 100 + 300 + 400 = 800 (the size of all four requests is 1100 what is greater than maximum allowed batch size).

**The default value of this options is 65507 bytes and can be changed by adding::** BatchSize = 65507

### 2.2.6 Maximum batch delay

This option determines how long JPaxos will wait for new requests to be packed into single instance.

**The default value of this option is 10 ms and can be change using::** MaxBatchDelay = 20

### 2.2.7 Network protocol

It is also possible to choose protocol used to communicate between replicas. One may choose:

- TCP (default)
- UDP
- Generic - Uses UDP for small messages and TCP for larger messages

It is important to note that UDP protocol has message size restriction - messages must be smaller than the maximum allowed size of UDP packet (64KB or less, depending on the network). User must be careful with the size of client requests and of BatchSize so that this limit is not violated. Because of this limitations, user should choose TCP or Generic option.

If one chooses Generic, it is also recommended to set what is a ‘small’ and what is a ‘big’ message, by setting maximum allowed UDP packet size:

```
Network = Generic
MaxUDPPacketSize = 1000
```

In example above, all messages smaller than 1000 bytes will be sent using UDP and all others using TCP protocol.

### 2.2.8 Example file

Below is an example configuration file:

```
# Nodes configuration
process.0 = 192.168.1.5:2000:3000
process.1 = 192.168.1.6:2001:3001
process.2 = 192.168.1.7:2002:3002

# Crash model configuration
```

```
CrashModel = EpochSS
LogPath = jpaxos/stableStorage

# Batching configuration
WindowSize = 2
BatchSize = 65507
MaxBatchDelay = 10

# Network configuration
Network = TCP
```

## 2.3 Configuration classes

The user of JPaxos has to configure the system by creating instance of `Configuration` class. Instance of this class is required to create the `Replica` and `Client` (see *Application programming interface* chapter). The `Configuration` class either loads the setting from certain file, or the settings may be provided by instantiating.

`Configuration` contains nodes configuration (information about replicas - ids, hostnames and ports), and replica related options (batching size, window size, etc.). `Client` is using only nodes configuration (hostnames and ports) and ignores replica related options, while `Replica` is using all options from `Configuration`.

The user is responsible for providing correct configuration to replicas and clients. The configuration must be the same in every replica and client or else the system may fail.

### 2.3.1 `Configuration` class

The `Configuration` has three constructors available:

- `Configuration ()` Default constructor, loads configuration from `paxos.properties` file located in current working directory. The structure of configuration file is described in “Configuration file format” section.
- `Configuration (String fileName)` Loads configuration from file specified as constructor argument. The format of the file is described in “Configuration file format” section.
- `Configuration (List<PID> processess)` Loads configuration using only nodes configuration. The `PID` structure is described below. Replica related options are set to default.

### 2.3.2 `PID` class

Stores node configuration data (information about one replica):

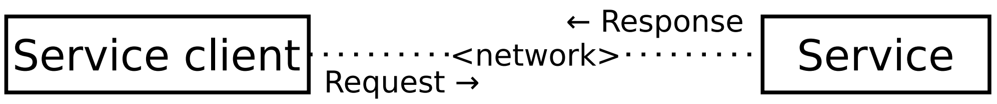
- `id` The id of replica. Ids start from 0.
- `hostname` The replica ip address (e.g. “127.0.0.1”) or host name (e.g. “localhost”). The replicas (and clients) will use this address to establish connection with this replica.
- `replicaPort` The port used by replicas to establish connection with this replica.
- `clientPort` The port used by clients to establish connection with this replica.

CHAPTER 3

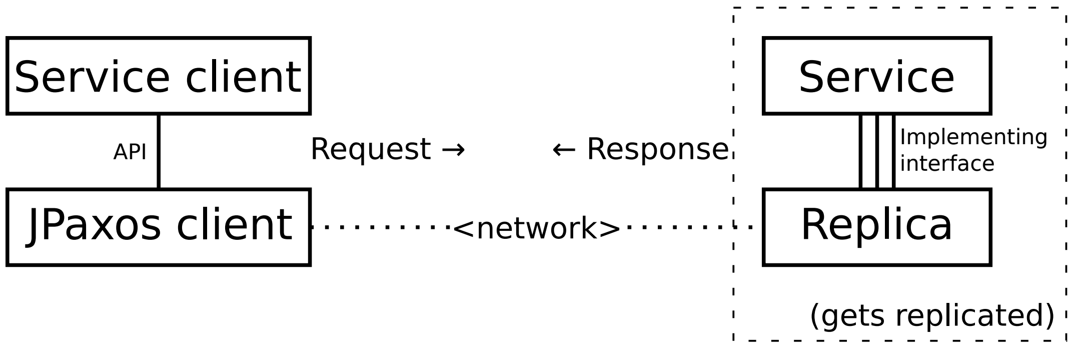
# Application programming interface

## 3.1 Introduction

A non-replicated service looks like this:



Our library replaces the `<network>` part. It looks like:



In order to make the replication and recovery possible, we require a few additional methods from the Service.

JPaxos provides implementation of it's Client and the Replica. The programmer must implement the Service and make use of the Client class.

`Client` sends requests to replicas and waits for reply. It provides only method for connecting and executing requests.

`Replica` uses Paxos algorithm to order client requests in all replicas; after deciding requests, executes them on service in proper order. All methods from `Service` class are called from one `Replica` thread, that means no two will be called concurrently on the service.

`Service` executes all requests from clients deterministically. The service must also implement a few additional methods to save and restore its state, which will explained in detail below.

Data exchanged by client and service are in form of byte arrays – `byte[]`. This gives biggest flexibility to the programmer - any data can be put there. One may use byte arrays in the service, as well as serialise/deserialise the `byte[]` to certain objects.

## 3.2 Service

JPaxos provides several base classes and one interface to create a new Service.

The `Service` interface, base for all services, describes methods that JPaxos requires from the service to be implemented. `Replica` calls all the API methods on the `Service` object. To launch the service, one needs to start the governing `Replica`.

The inheritance hierarchy is as follows:

```
interface Service
  abstract class AbstractService
    abstract class SimpleService
      abstract class SerializableService
```

Using the `Service` interface is discouraged in favour of `AbstractService`, as the latter implements methods common to all services.

`AbstractService` provides widest range of functionality. `SimpleService` is the `AbstractService` narrowed to the absolute minimum for the system to work. `SerializableService` differs from `SimpleService` only with data type - all client requests and service responses are serialised using Java serialisation.

### 3.2.1 `SerializableService` class

This is the simplest class of all available.

This class has minimal number of methods. One requires from the `SerializableService` to be able to execute client requests, be able to create a snapshot and roll back to state from a snapshot.

`SerializableService` uses Java serialization for creating byte arrays from Object, therefore snapshot, request and reply classes must be serialisable.

Method summary:

- `abstract Object execute (Object value)`

  Executes a command from client on this state machine. The return value is sent back to the client.

- `abstract Object makeSnapshot ()`

  Makes snapshot of current Service state. The same data created in this method, will be used to update state from snapshot using `updateToSnapshot(Object)` method.

- `abstract void updateToSnapshot (Object snapshot)`

  Updates the current state of Service to state from snapshot. This method will be called after recovery to restore previous state, or if we received newer snapshot from other replica (using catch-up).

### 3.2.2 `SimplifiedService` class

This class also has minimal number of methods.

Only difference between `SimplifiedService` and `SerializableService` is the type of passing the data - the `SimplifiedService` uses byte arrays for that.

Method summary:

- `abstract byte[] execute (byte[] value)`
- `abstract byte[] makeSnapshot ()`
- `abstract void updateToSnapshot (byte[] snapshot)`

### 3.2.3 `AbstractService` class

This class supports three additional functionalities in comparison to `SimplifiedService` and `SerializableService`:

- service may choose when the snapshot has to be done
- service may know if the recovery has finished
- snapshots can be passed for state after specific request, not only after the last request

These features let the service be more flexible and make using threads inside the service code easier. This comes at cost of placing partial responsibility on the service programmer - he must provide valid arguments for the functions.

In order to preserve the functionality, some additional data needs to be remembered by the service. A sequential number of request identifies each request and request order in JPaxos - these numbers must be taken under consideration in the service as well. This sequential number is global among the replicas.

To make the snapshot creation easier, JPaxos checks the size of logs and previous snapshots. If the ratio between these sizes reaches certain levels (see Configuration chapter) methods `askForSnapshot` and `forceSnapshot` are called. Programmer should use these methods as hints, although one may just make the snapshot then, or ignore these methods at all.

Method summary:

- `abstract byte[] execute (byte[] value, int executeSeqNo)`

  Executes the request (`value`) and returns the reply for client. The return value must not be null. The `executeSeqNo` is the sequential number of this request.

- `abstract void askForSnapshot (int lastSnapshotNextRequestSeqNo)`

  Notifies the service that it would be good to create a snapshot now.

- `abstract void forceSnapshot (int lastSnapshotNextRequestSeqNo)`

  Notifies the service that log is very large and a snapshot should be made.

- `final protected void fireSnapshotMade(int nextRequestSeqNo, byte[] snapshot, byte[] response)`

  Informs the replica that new snapshot has been made and passes it as the byte array `snapshot`. `nextRequestSeqNo` is the sequential number of first request that will be executed after the snapshot.

If this method is called from within execute method, after the just executed request, the `response` must be also provided (otherwise may be null).

- `abstract void updateToSnapshot (int requestSeqNo, byte[] snapshot)`

  Updates the current state of `Service` to the state from the snapshot.

- `void recoveryFinished()`

  Informs the service that the replica is fully functional - i.e. recovery process has been finished (if any).

  This implementation is an empty method. One may not override it, if the knowledge that recovery has finished is not needed.

### 3.2.4 `Service` interface

The service interface describes methods that JPaxos requires from the service to be implemented.

Some methods in this interface use `SnapshotListener` interface. This interface is very simple, it contains one method only: `void onSnapshotMade(int requestSeqNo, byte[] snapshot, byte[] response)`. This method must be called by `Service` when a new snapshot has been made on all previously registered listeners. The parameters match exactly the ones from `fireSnapshotMade` method from `AbstractService` class.

Method summary:

- `byte[] execute (byte[] value, int executeSeqNo)`
- `void askForSnapshot (int lastSnapshotNextRequestSeqNo)`
- `void forceSnapshot (int lastSnapshotNextRequestSeqNo)`
- `void updateToSnapshot (int requestSeqNo, byte[] snapshot)`
- `void addSnapshotListener (SnapshotListener listener)`

  Registers a new listener. Each listener has to be informed every time a snapshot has been created by `Service`.

- `void removeSnapshotListener (SnapshotListener listener)`

  Unregisters the listener.

- `void recoveryFinished()`

## 3.3 Replica

In order to achieve replication above the Paxos protocol another layer must be implemented - a layer that passes the Paxos decisions to the service and accepts client requests. This part is called Replica. Each Replica has one copy of underlying `Service`. A replica must have it's unique number - called local Id, or just Id. The Id's are sequential numbers starting from 0.

To create the `Replica` object you need:

- configuration (shared by replicas and clients)
- local Id - identification number

- the `Service` that has to be replicated

The constructor and methods from Replica class are described below:

- `Replica(Configuration config, int localId, Service service) throws IOException`

  Creates the replica with given `Service` and specified Id.

  The `Configuration` class is described in the *Configuration class* chapter

- `void setLogPath(String path)`

  Sets path for the logs used by certain crash models. The location of logs must be different for each replica. The path for the logs may also be set in configuration file. However, this method overrides the configuration file setting.

  If the log path is set neither in configuration file nor using this method, default `jpaxosLogs/<id>` location is used.

- `void start() throws IOException`

  Starts the replica – i.e. starts the recovery process and subsequently launches the `Service`.

## 3.4 Client

Creating a client is very simple – each client needs only replica configuration. Then you can easily connect to replicas, and execute commands on them. `Client` class takes care of all details related with reconnecting if replica crashes and sending command to other ones.

The Client class has the following methods:

- **`Client() throws IOException`** `Client(Configuration config) throws IOException`

  Creates the client. The `Configuration` should be the same as for Replica. The first version reads default paxos.properties file, while the latter lets the user choose file location.

- `void connect()`

  Connects to the replicates service. This method blocks until the connection will be established.

- `synchronized byte[] execute(byte[] request)`

  Executes the request and waits for the response.

**SerializableClient**

If one chose using `SerializableService` as base class for the service implementation, one should use class `SerializableClient` rather than `Client`. The only difference is that this class deserialises the responses back to objects.

CHAPTER 4

# Example: replicated hash map

This example presents simple service implementation. The service is a replicated hash map. Each client command reads requested entry and modifies it. Service responds with old map entry value.

## 4.1 Service

First, we present an example Service imlementation.

```java
import lsr.service.SimplifiedService;

public class SimplifiedMapService extends SimplifiedService {

    // Map to be replicated
    private HashMap<Long, Long> map = new HashMap<Long, Long>();

    /** Processes client request and returns the reply for client **/
    @Override
    protected byte[] execute(byte[] value) throws IOException {

        // Deserialise the client command
        MapServiceCommand command;
        command = new MapServiceCommand(value); // this class is not included
        if(!command.isValid())                  // in the example
            return new byte[0];

        // We do the work
        Long oldValue = map.get(command.getKey());
        if (oldValue == null)
            oldValue = new Long(0);
        map.put(command.getKey(), command.getNewValue());

        // We serialise the message back
        ByteArrayOutputStream byteArrayOutput = new ByteArrayOutputStream();
        DataOutputStream dataOutput = new DataOutputStream(byteArrayOutput);
        dataOutput.writeLong(oldValue);

        // And return the reply to the client
        return byteArrayOutput.toByteArray();
    }
    /** Makes snapshot used for recovery and replicas that have very old state **/
    @Override
    protected byte[] makeSnapshot() {
        // In order to make the snapshot, we just serialise the map
        ByteArrayOutputStream stream = new ByteArrayOutputStream();
        try {
            ObjectOutputStream objectOutputStream = new ObjectOutputStream(stream);
            objectOutputStream.writeObject(map);
        } catch (IOException e) {
            throw new RuntimeException("Snapshot creation error");
        }
        return stream.toByteArray();
    }

    /** Brings the system up-to-date from a snapshot **/
    @Override
    protected void updateToSnapshot(byte[] snapshot) {
        // For map service the "recovery" is just recreation of underlaying map
        ByteArrayInputStream stream = new ByteArrayInputStream(snapshot);
        ObjectInputStream objectInputStream;
        try {
            objectInputStream = new ObjectInputStream(stream);
            map = (HashMap<Long, Long>) objectInputStream.readObject();
        } catch (Exception e) {
            throw new RuntimeException("Snapshot read error");
        }
    }
}
```

## 4.2 Replica

In order to run the service, one needs also to write an application starting the service.

```
public static void main(String[] args) throws IOException {

    /** First, we acquire the ReplicaID **/
    if (args.length > 2) {
        System.exit(1);
    }
    int localId = Integer.parseInt(args[0]);

    /** Then we create the replica, passing to it the service **/
    Replica replica = new Replica(new Configuration(), localId, new SimplifiedMapService());

    /** Then we start the replica **/
    replica.start();

    /** And the service runs until the enter key is triggered **/
    System.in.read();
    System.exit(0);
}
```

## 4.3 Client

The code below presents the client side.

```java
import lsr.paxos.client.Client;

/** Creating the Client object **/
Client client = new Client();
client.connect();

(...)

/** Prepairing request **/
MapServiceCommand command = new MapServiceCommand(key, newValue);
byte[] request = command.toByteArray();

/** Executing the request **/
byte[] response = client.execute(request);

/** Deserialising answer **/
DataInputStream in = new DataInputStream(new ByteArrayInputStream(response));

System.out.println(in.readLong());
```

# Bibliography

- *The part-time parliament*. L. Lamport. ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169
- *Paxos for System Builders: an overview*. Kirsch, Jonathan and Amir, Yair. LADIS '08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware
- *Paxos mad live: an engineering perspective*. Chandra, Tushar D. and Griesemer, Robert and Redstone, Joshua. PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing

# Index