12-2019

JeroMF: A Software Development Framework for Building Distributed Applications Based on Microservices and JeroMQ

Aditi Jain
Utah State University

JEROMF: A SOFTWARE DEVELOPMENT FRAMEWORK FOR BUILDING DISTRIBUTED APPLICATIONS BASED ON MICROSERVICES AND JEROMQ

by

Aditi Jain

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

_____	_____
Stephen Clyde, PhD	Curtis Dyreson, PhD
Major Professor	Committee Member

_____
Vladimir Kulyukin, PhD
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2019

ABSTRACT

JeroMF: A Software Development Framework for Building Distributed Applications Based on Microservices and JeroMQ

by

Aditi Jain

Utah State University, 2019

Major Professor: Stephen Clyde, PhD
Department: Computer Science

This report describes a project involving the design, implementation, and testing of a software development framework, called JeroMF, that can help developers create scalable distributed applications based on a microservice architecture and that uses JeroMQ (a native Java implementation of ZeroMQ) for message passing. JeroMF provides an execution framework and extensible components for implementing processes, services, communication channels, messages, communication statistics, and encryption. Applications built with JeroMF do not require a message broker or any other middleware processes. However, they may include an optional *Service Registry* that can facilitate for service discovery and secure communications. The Service Registry itself was implemented with JeroMF and is included as part of the JeroMF distribution. Thorough unit, integration and system test cases exist for every component of JeroMF. For validation, JeroMF was used to re-design and re-implement a distributed health-care application with 13 separate types of services and very strict security requirements.

(43 pages)

ACKNOWLEDGMENTS

I would like to thank my major advisor, Dr. Stephen Clyde. His invaluable advice, guidance and encouragement throughout my graduate level education enabled me to develop an understanding of the subject. His suggestions, support and hard work are deeply appreciated.

I would especially like to acknowledge my gratitude to my committee members, Dr. Vladimir Kulyukin, and Dr. Curtis Dyreson, for their time and attention throughout the entire process.

Many thanks to Genie Hanson, Cora Price and Vicki Anderson their help were indispensable for completing my graduation.

Finally, I give special thanks to my parents, my sister and friends for their encouragement and moral support as I worked my way through the duration of my academic pursuits. I could not have done it without all of you.

Aditi Jain

CONTENTS

Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Most modern distributed applications need to be scalable, as well as extensible, and flexible. Some even need to allow for dynamic service composition, which is the aggregation of existing services at runtime to fulfilling emerging needs [1]. To fulfill these needs, researchers have created and published many libraries and frameworks for distributed applications based on a service-oriented architecture. However, a service-oriented architecture can be unwieldy, particularly when faced with massive scalability requirements. Moreover, most service-oriented architectures require services to communicate via enterprise service bus (ESB), which could become a single point of failure [2]. In contrast, applications based on microservice architectures do not require an ESB. Also, services in microservice architecture can be operated and deployed independent of each other, allowing for more flexible and frequent deployments and scalability [2].

Microservices are an architectural style for structuring applications around loosely coupled services and for making those services as granular as possible without compromising efficiency [3]. Microservices are highly maintainable, testable, independently deployable, and scalable [4]. Also, software engineers can, organize them around business capabilities and thereby creating systems with excellent modularity and encapsulation, which help to enable dynamic service composition and improve overall reliability, security, and fault tolerance. Microservices also enable the continuous deployment of large, complex applications. Section 2.1 provides more details on microservices.

However, without a development framework, an application based on microservices can be hard to construct, test, debug, deploy, and maintain. Simply, splitting an application into multiple independent services generates more artifacts to manage without necessarily obtaining the desirable properties mentioned above. In fact, a haphazard refactoring of a distributed application into lots of services could create more complicated deployment and operational procedures, since the application has more independent parts that need to be launched on host machines and that need to communicate with each other.

When building an application based on microservices, developer need to handle requests between services carefully, managing database connections within the context of individual services, and avoid potential data integrity problems caused by the failure of transactions involving multiple services.

In general, software development frameworks are libraries of reusable components with execution infrastructures [5]. The execution-infrastructure portion of a framework implements “inversion of control”, so developers don’t have to write the execution or control logic directly and can focus on the behaviors that are unique to the application [5]. Although this might constrain developers with respect to architectural choices, it can help to make development more productive when building applications with architectures that align with what the framework supports. There are many software development frameworks available today, such as .Net for Windows development and Cocoa for Mac OS X.

In 2007, Pieter Hintjens along with Martin Sustrik introduced ZeroMQ as a high-performance, asynchronous, lightweight messaging library for scalable distributed

applications [6]. ZeroMQ is fast, simple, reliable and provides easy scalability. Also, has been ported to over 40 different programming languages, including a native implementation for Java, called JeroMQ. The API for ZeroMQ and JeroMQ are the same for in-process, inter-process, peer-to-peer, and multicast communications. See Section 2.2 for more details on ZeroMQ.

Developers working with ZeroMQ can create distributed application more quickly and with higher quality than with lower-level socket libraries because it is very simple to use. It handles all the socket details, enabling developers to focus on building inter-process communications rather than socket interactions.

However, ZeroMQ is just a class library and not a development framework. As such, it does not directly support any architecture. So, even though, it is an excellent choice for implementing the communications of microservices, it does not help with the applications based on the Microservice Architecture style. Specifically, developers using ZeroMQ may still have to address the following important challenges:

- Defining microservices: This includes implementing the whole life cycle of a microservice from conception to maintenance.
- Defining efficient and secure communication: Microservices run in separate processes and communicate to use each other, typically over an open network. Developers need to create resilient microservices that can withstand situations like failure of other processes, failure of a database service or database connection, lost messages, and hacker attacks.
- Maintaining transaction safety and consistency: ACID (atomicity, consistency, isolation, and durability) is an acronym for four transaction properties that can

ensure consistency and validity in data systems. However, with microservices, transactions can span over multiple services, making it difficult to achieve consistency during situations like transaction roll back [7].

- Allowing for service discovery: Three major challenges in any distributed system is discovering where shared resources exists, who manages them, and what can be done with those resources. For service-oriented and microservice architectures, these challenges all relate to service discovery [8]. So, when building a microservice application, developers need a service registry that allows micro-service to find each other.
- Refactoring existing logic into services: When moving existing application code into microservices, developers may need to untangle convoluted dependencies so each service can be cohesive and loosely coupled.

This report describes an open-source software development framework, called "JeroMF", for creating dynamic systems based on microservice architectures with JeroMQ as the communication framework. Its goal is to make it easier for developers to create secure and reliable distributed applications by providing an execution framework and reusable components for processes, services, communication channels, messages, and communication statistics. Chapter 3 provides a detailed explanation of JeroMF's design, along with snippets of its implementation. The full implementation is available as an open source project in a Git [9] repository.

To verify JeroMF, we have created executable unit, integration, and system test cases. These test cases provide reasonably thorough coverage using path, input-

validation, and logic-based testing techniques. See Chapter 4, and specifically Sections 4.2-4.3, for more details on unit, integration, and system testing.

Validating JeroMF requires using it to implement a real-world distributed application. For this, I selected the Synchronization Facility of the Utah Department of Health's Child Health Advanced Records Management (CHARM) system as a test case for JeroMF. For details, see Section 4.4.

A literature review has not uncovered any other frameworks for developing microservice applications using JeroMQ, but some researchers have addressed problems like those addressed by this research. Chapter 5 summarizes related literature and discusses their contributions relative to JeroMF.

An initial prototype of "JeroMF" has been created and is currently being integrated into several CHARM components.

CHAPTER 2

BACKGROUND

2.1. Microservices

The Microservice Architecture style is an approach to developing applications as sets of small independent services[1]. Each of the services runs independent of all other services and encapsulates its logic behind a network-accessible API. Ideally, a microservice should be able to be developed independent of other microservices, except for knowledge of their API's. To complete the work of a distributed application, a microservice may engage microservices for parts of that work. Any centralized management of these services, if needed, can be simply be implemented as another service. Services may be written in different programming languages, as long as the communication protocols are clearly defined.

This architectural style approach has grown popular in recent years as enterprises look to become more agile and move towards continuous testing, integration, deployment, and delivery. Also, because of the abstraction, modularity, and encapsulation that it encourages, the microservice-architectural style helps with creating scalable and testable distributed applications.

Microservices are loosely coupled, meaning that each application component has little to no knowledge of the definitions of the other components. Each service uses its own database and functions independently of the larger application.

Microservices have smaller code bases that are easy to maintain and more fault tolerant. If some of the services go down, only some features will be affected and not the entire application [2]. They can be independently developed, deployed, separately

versioned, and reused [3]. They can be scaled independently, making it easy to increase the availability of services that have become bottlenecks, while leaving less frequently used services untouched [4]. Moreover, different programming languages and runtime platforms can be used for each service, since services run independently of one another [1].

These microservices have smart end points and dumb pipes through which they receive requests, process them and generate response accordingly [5]. For communications, microservices uses either synchronous protocols or asynchronous protocols over transport protocols, like TCP and UDP, or other application-level protocols, like HTTP.

Microservices provides several benefits relative to the challenges that developers may come across while building distributed applications, including development, testing complexities, dynamic network topologies, and issues with consistency across services and transactions [2].

2.2. ZeroMQ

ZeroMQ (also spelled ØMQ, 0MQ or ZMQ) is a high-performance asynchronous messaging library, aimed for scalable distributed or concurrent applications[6]. Pieter Hintjens states in ZeroMQ guide that “ZeroMQ looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous

message-processing tasks. It has a score of language APIs and runs on most operating systems” [7].

ZeroMQ is designed for efficiency, even for extremely high message volume. Other messaging systems rely on a central message broker through which all messages pass. ZeroMQ follows a broker-less design and, therefore, does not introduce a single point of failure into a distribution application [8].

It handles all the socket implementation, hiding complexities of TCP connection and buffering, and thus increasing developer’s productivity to focus on system requirements rather than on socket manipulation or network failures.

It can handle thousands of concurrent connections, while ensuring the integrity of data and without unreasonably affecting the speed of the operation. Even if a connection is lost or process fails, messages are not lost. They will remain in a queue until the disconnected clients are reconnected and will be delivered in order [10].

ZeroMQ also provides for basic communication patterns, like publish/subscribe" and "request/reply". Each communication pattern is orthogonal to other patterns and can be thought of as a separate tool [8]. Moreover, ZeroMQ takes advantage of multiple cores, if they are available, and thereby can provide performance improvements without developers having to explicitly address that issue[8].

I will use JeroMQ (ZeroMQ’s native implementation in Java) as the messaging framework for JeroMF.

CHAPTER 3

DESIGN AND IMPLEMENTATION OF JEROMF

3.1 Architectural Design Overview

This chapter explains the architectural design of JeroMF, which provides both an execution framework and collection of extensible components for building distributed applications based on microservices.

The UML Class Diagram [11] in Figure 3.1 shows JeroMF's primary packages with their essential classes and relations. From left to right are the base components for implementing custom processes, application-specific services, and communications. Developers create distributed system in JeroMF by implementing specializations of these components or by reusing them directly.

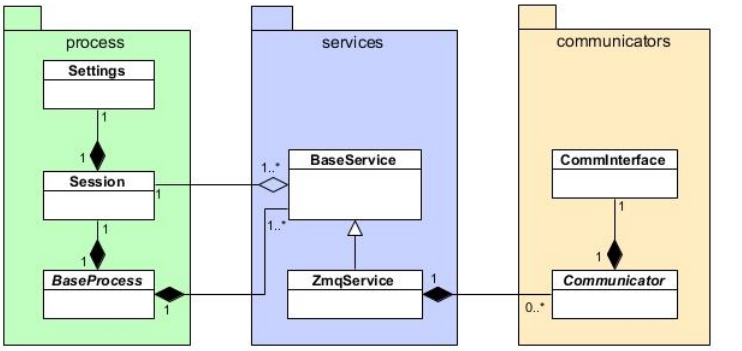

Figure 3.1 – The primary packages in JeroMF with their key classes and relationships.

To illustrate the architecture and use of JeroMF, we use a simple distributed application for managing used cars for multiple dealers (see Figure 3.2). With this sample application, every used-car dealer would run its own Used-car Server (only one shown in Figure 3.2) and each Used-car Server would contain a microservice,

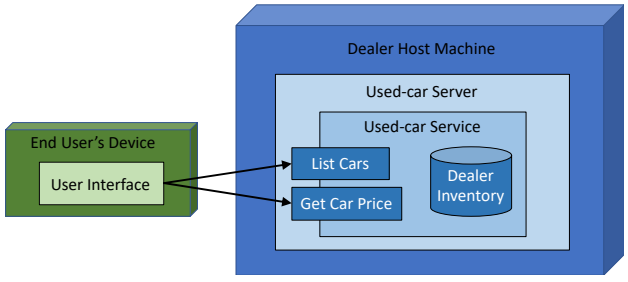

Figure 3.2 - Sample Distributed Application for Tracking Used Cars

called Used-car Service. This service would encapsulate the dealer's own used-car data and provide a network-accessible API that would allow remote clients, e.g., the end user interface, to query what cars the dealer currently has in inventory and their prices. This sample application is minimal and only for illustration purposes. It does not contain all the functionalities one would expect in a real used-car application.

The following sections describe JeroMF's components in more detail, beginning with the process-related components.

3.2 Process

Figure 3.3 illustrates the key components of the Process package in more detail. A process in JeroMF, defined at least in part by the BaseProcess class, is an execution container that holds one or more services. Its implementation must be a specialization of BaseProcess, which contains a single Session object and in turn the Session object contains a single Settings object.

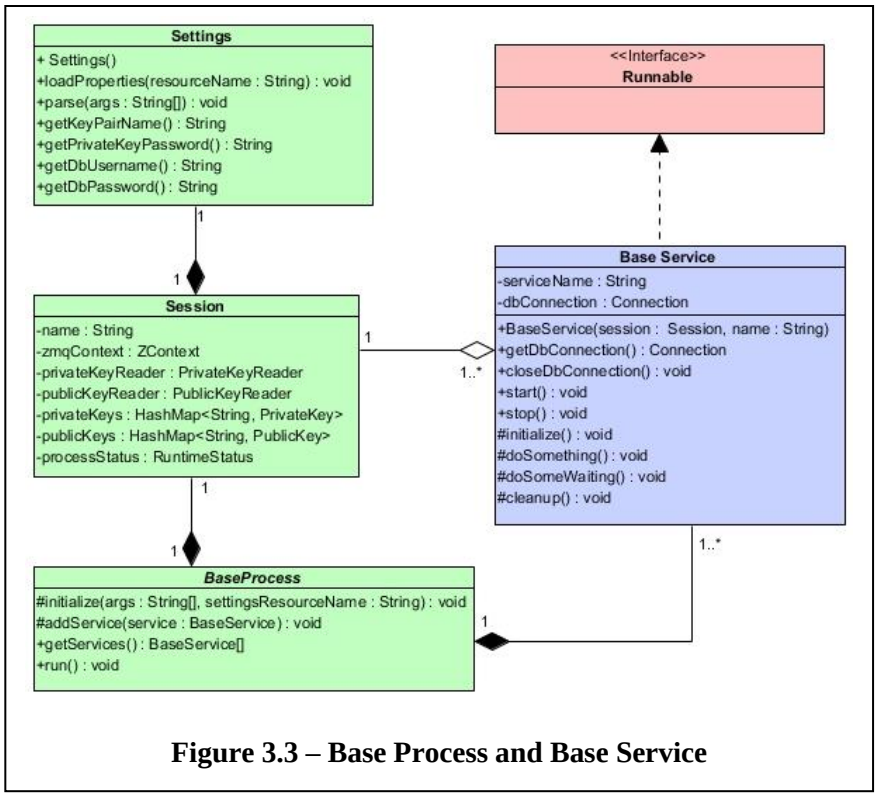

Figure 3.3 – Base Process and Base Service

If a developer is following a strict microservice architecture, then each JeroMF

process will hold exactly one service. However, JeroMF allows a process to hold more than one, at the developer's discretion, to achieve certain execution and development efficiencies in certain situations.

Developers will implement application-specific processes by specializing the BaseProcess class and then implementing the main() method (see Figure 3.4), as well as any desired overrides to the following methods: initialize(), validateSettings(), createSettings(), createSession(), and cleanup().

```
public class MyProcess extends BaseProcess {
  public static void main(String[] args){
    MyProcess instance=new MyProcess();
    try{
      instance.initialize(args,"my.properties");
      // TODO: instantiate services
      // TODO: add them to instance
      instance.run();
    } catch (Exception e) {
      e.printStackTrace();
    }
    finally {
      instance.cleanup();
    }
  }
  …
```

Figure 3.4 – Template for the main() method

See Figure 3.5 for a partial UML class diagram of used-car application with the components implemented by developer in brown.

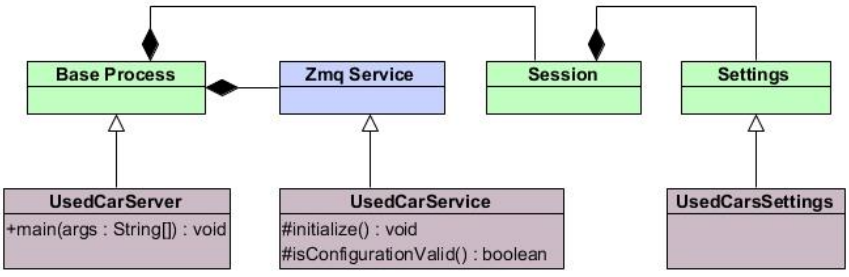

Figure 3.5 – Used car example

Figure 3.6 shows the code for UsedCarServer process. Note how the implementation of main() follows the template shown in Figure 3.4. When the program runs, it first creates an instance of UsedCarServer. (Section 3.6 will provide more details on UsedCarService and its base class, ZmqService.) It then sets up that instance by

```
public class UsedCarServer extends BaseProcess {

    public static void main(String[] args){
        UsedCarServer instance=new UsedCarServer();
        try{
            instance.initialize(args,"server.properties");
            UsedCarService usedCarService=
                 new UsedCarService(instance.getSession(),"UsedCarsService");
            instance.addService(usedCarService);
            instance.run();

        } catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            instance.cleanup();
        }
    }

    @Override
    protected Settings createSettings() {
        return new UsedCarSettings();
    }
}
```

Figure 3.6 - UsedCarServer

calling the initialize() method. The initialize() method is an application of the Template Method pattern [12] that 1) creates a Session object by calling createSession(), 2) creates Settings objects by calling createSettings(), and 3) validates the Settings object by calling validateSettings().

A developer may override the initialize() method, but typically does not need to. However, if a developer does override of the initialize(), then the override should invoke super.initialize() to make sure that Session and Settings objects are created and validated properly.

After calling initialize(), the main() method creates all the necessary services (just one is the use-car example), and add them to the process instance. Then, main() calls the run() method on the instance, which will starts up the services. The run() method will not return until all contained services stop. The rest of code in main() is for error handling.

3.3 Session

In general, Session object keeps track of the process's name, Settings object, status, JeroMQ context (ZContext), and encryption keys. Session object is shared with all services that the process creates so they can access the session information. This is done by passing the session object as the first parameter to service's constructor. For example, see the call to the UsedCarService constructor in Figure 3.6.

If the distributed application needs a custom Session object, the developer simply derives a specialization from Session and overrides the createSession() method in the process class to return an instance of that specialization.

As mentioned, a Session object contains the process's status. The possible values for this status are as follows: Created, Initializing, Initialized, Running, Stopping, Stopped, and Failed. They represent the states in the lifecycle of BaseProcess (see Figure 3.7). In general, a process's state progresses from Created to Stopped, unless there is an unrecoverable failure, in which case the projects transitions to the "Failed" state.

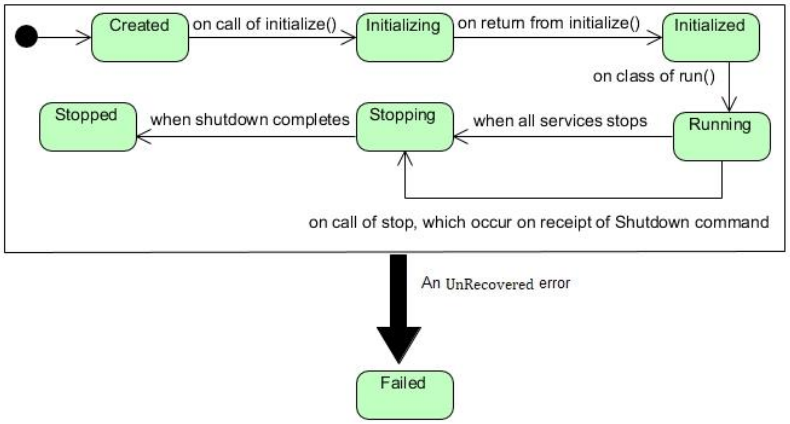

Figure 3.7 – A UML State Chart describing the general behavior of BaseProcess objects.

On construction, a Session object will create and manage a ZContext object, which is what JeroMQ uses to manage the underlying communications. Developers don't have to worry about creating or cleaning

this object, because the Session object handles this automatically.

3.4 Settings

As mentioned above, the process's initialize() method creates a Settings object and stores it in the Session object. This Settings object holds all the configurable settings for a process. Each setting has default value that can be changed through properties files, environment variables, or command-line parameters. Appendix A.1 lists of all the settings defined by the Settings base class and their default values and explains how they can be configured.

One of the most important settings, called "private-key-password", contains the password for unlocking a private key used for asymmetric encryption. To avoid security leaks, a value for this setting should never be included in a property file, which are often stored in source code repositories or otherwise distributed. This and other confidential settings should only be configured using an environment variable or command-line parameter.

Like the Session class, the Settings class can be specialized to meet the specific needs of a distributed application. To do so, a developer, simply defines a class that derives from Settings class, adds the properties and methods that store and parse the custom settings, and then implements an override for the createSettings() method in the process class to return an instance of the specialization. For example in Figure 3.6, the UsedCarProcess needs a custom Settings, so developer implements a specialization of and then overrides createSettings() method to return instance of that specialization. See Appendix C for the implementation of the UsedCarSettings class that is referenced in createSettings() override in Figure 3.6.

3.5 Services

The BaseService class (see Figure 3.3) represents a microservice with an optional database connection for persistent shared resources. It has access to the process's Session object, which was provided as a parameter to the service's constructor.

To start a service, the process calls the start() method and to stop it, the process calls stop() method. Since the process's run() method managing service execution, application developers should not have to work with these methods directly.

Figure 3.8 shows the implementation of the run() method of the BaseService class. This method is executed when the start() method is invoked and, like the run method for BaseProcess, is an application of Template Method pattern. The customizable parts of the method are encapsulated in the initialize(), doSomething(), doSomeWaiting(), and cleanup() methods. The initialize() method is called once at the

```
@Override
public void run() {
  if (processSession ==null) return;
  serviceStatus = RuntimeStatus.Initializing;
  try {
    initialize();
    serviceStatus = RuntimeStatus.Running;
    while (serviceStatus == RuntimeStatus.Running &&
           processSession.getProcessStatus() == RuntimeStatus.Running) {

      doSomething();
      if ((serviceStatus ==RuntimeStatus.Initializing ||
           serviceStatus == RuntimeStatus.Running) &&
           processSession.getProcessStatus() == RuntimeStatus.Running) {
        doSomeWaiting();
      }
    }
  }
  catch (ServiceException e) {
    stop();
  }
  if (serviceStatus == RuntimeStatus.Running) {
    stop();
  }
  cleanup();
  serviceStatus = RuntimeStatus.Stopped;
}
```

Figure 3.8 - The run() method of BaseService class

beginning and is intended to setup the resources that service will need to do its work. The doSomething() method implements the work of service. The doSomeWaiting() method should first check to see if there is any more work to be done. If not, it should wait for more work to become available. If even there is some work to be done, it may gracefully give up the CPU so other processes have a fair chance to run. The cleanup() method releases resource allocated by initialize() or during the execution of work performed by doSomething(). The BaseProcess class contains default implementations for all these methods, so specializations only has to provide overrides where functionality needs to be different from the default behavior.

Every service has its own status, independent of the process but with the same possible values as the process's status. Through the session object, a service can also access the process's overall status. If the process's status ever becomes something other than Running, then a service will gracefully shutdown itself down. See the loop and if conditions in code shown in Figure 3.8.

3.6 ZmqService

The **ZmqService** class is a specialization of BaseService class that represents a microservice with communication capabilities based on JeroMQ (see Figure 3.9). As such, it can have zero or more communicators, i.e., instances of the **Communicator** class, for interacting with other services or clients.

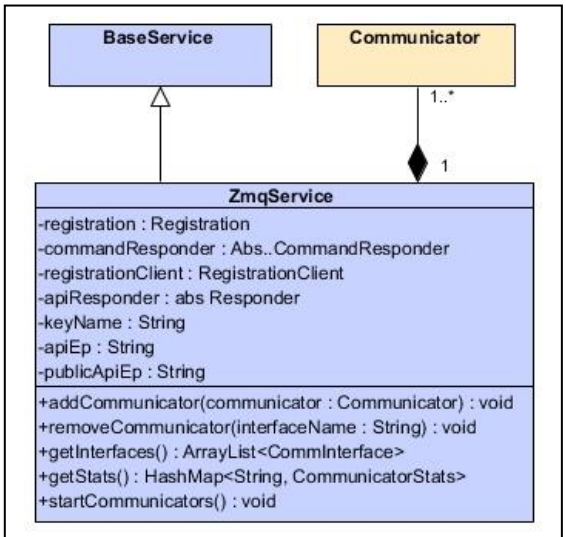

Fig 3.9 - ZmqService

Typically, a ZmqService would include three specialized Communicators

- A RegistrationClient that is responsible for listing the service with the Registry and to setup a shared secret key for encrypted communications.
- A CommandResponder that listens for general control messages from the Registry or some other control process.
- An APIResponder for handling requests from clients.

None of these communicators are required and are only setup if their end-point settings have values in the Settings object.

Although BaseService and ZmqService can be used as it is for instantiating many types of services, they can further be customized through specialization. Like JeroMF process, services have initialize() and run() methods that follow the Template Method pattern, with the customizable parts encapsulated in virtual methods.

The Specializations of ZmqService may override any of the methods defined in ZmqService, but the overrides methods should typically call ZmqService's implementations somewhere in their code. Most specialization of ZmqService only need to override initialize() and doSomething(). In these methods, a specialization of ZmqService can do one or more of the following:

Adding and removing other communicators

Specializations of the ZmqService class can add other Communicators by simply instantiating them and calling addCommunicator() method. Typically, this would be done in an override of the initialize() method. To remove a Communicator from its list of Communicators, it should call removeCommunicator() method. This would typically be done in an

override of the stop() method.

Allocation other resources besides a database connection

Specialization of ZmqService can allocate other resources. This would be done in an override of the initialize() method, after calling its super's (i.e., ZmqService's) initialize() method; it would set up and allocate other resources that is needed for working.

Perform custom work, independent of incoming requests

Specialization of ZmqService would perform its work in an override of the doSomething() method.

See Figure 3.10 for implementation of a simple microservice for used car example. UsedCarService is specialization of the ZmqService for the used-car application. It overrides the initialize() method for customizing message handlers of its APIResponder. The initialize() method calls its super's (ZmqService's) initialize() method, which sets up the instances of the three types of communicators. ZmqService's initialize() method also calls its super's (i.e., BaseService's) initialize() method, which sets up everything that is needed for working with the database.

The UsedCarService's initialize() method after calling super's initialize() method customizes its APIResponder, provided by zmqService to handle two types of messages, namely ListCars and GetCarPrice by adding message handlers for them. A message handler for a type of message defines what kind of encryption to expect for the incoming message and what type of encryption to use for the reply, along with a lambda function for processing incoming messages. In this example, the both lambda function simply call private methods.

```
public class UsedCarService extends ZmqService {
  UsedCarService(Session session, String serviceName)
  throws ServiceException {
      super(session, serviceName);
 }

@Override
protected void initialize() throws ServiceException {
  super.initialize();
  apiResponder.addMessageHandler(ListCars.class, EncryptionMode.None,
               EncryptionMode.None, message -> listCars());
  apiResponder.addMessageHandler(GetPriceForCar.class,
               EncryptionMode.None, EncryptionMode.None,
               message -> getPriceForCar((GetPriceForCar) message));
}

private Message listCars(){
  Connection conn = null;
  try{ conn = getDbConnection(); } catch(ServiceException se){…}
  if (conn==null) return null;
  String usedCarsLoadSql = "SELECT * FROM usedCars ";
  try {
    ArrayList<UsedCar> usedCars = new ArrayList<UsedCar>();
    ResultSet rs=conn.createStatement().executeQuery(usedCarsLoadSql);
    while(rs.next()
      userCars.add(new UsedCar(rs.getInt("id"),
                           rs.getInt("year"),
                           rs.getInt("make"),
                           rs.getString("model")));
    Message listCars=new ListCarsReply(usedCars);
    return listCars;
  } catch (SQLException e) {}
  return null;
}

 private Message getPriceForCar(GetPriceForCar message){
   Connection conn = null;
   try { conn = getDbConnection(); } catch(ServiceException se){…}
   if (conn==null) return null;

   String getPriceSql = "SELECT price FROM usedCars WHERE "+"id=@id";
   NamedPreparedStatement statement = null;
   try {
    statement = new NamedPreparedStatement(conn,getPriceSql);
    statement.setInt("id",message.getCarId());
    ResultSet resultSet=statement.executeQuery();
    double price=0;
    if(resultSet.next()){
       price=resultSet.getDouble("price");}

    Message carPrice = new CarPrice(message.getCarId(),price);
    return carPrice;
  } catch (SQLException e) {…}
  return null;
}
```

Figure 3.10 Code snippet of UsedCarService

The private methods listCars() and getPriceForCar() are invoked when ListCar

message and GetCarPrice message respectively are received by the APIResponder. These methods get a reference to database connection using a protected method inherited from BaseService and then use that connection to retrieve the requested information. Then they create reply messages ListCarReply message for ListCar message and CarPrice reply message for GetPriceForCar message. If the desired information could not be retrieved these methods return null as message reply.

3.7 Communicators

3.7.1 Overview

The Communicator is an abstract class for the objects that handle all the communications in JeroMF. They use JeroMQ, which turn in turn uses one of three transport-layer communication mechanisms, namely: TCP, inproc, or ipc [7]. Each communicator has an end point that defines both the transport-layer communication mechanism and either the local

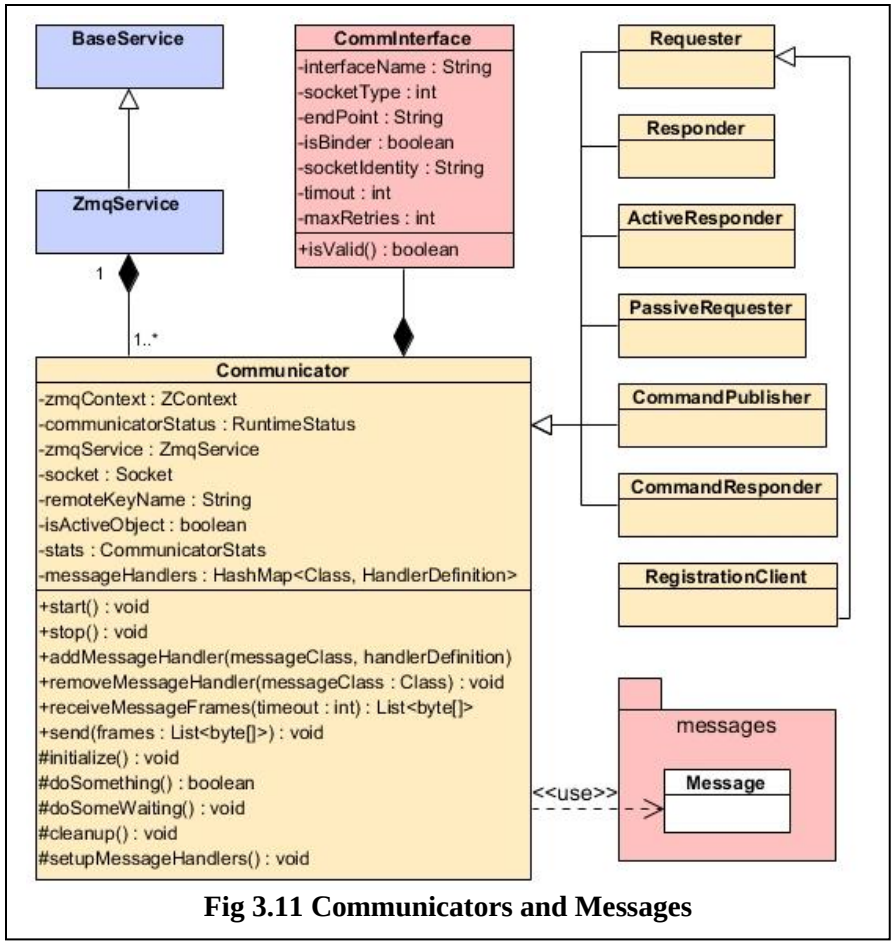

Fig 3.11 Communicators and Messages

address that the communicator will bind to or the remote address it will connect to. The details about a communicator's end point are defined by an instance of CommInterface class. Developers do not need to directly create or access these objects.

JeroMF includes six reusable communicators (see Figure-3.11). The Requester and Responder Communicators handle reliable request-reply style communications where the requester initiates all conversations (see Section 3.9.1). The Active Responder and Passive Requester also handle reliable request-reply style communications, but the responder starts by indicating its readiness to receive requests (see Section 3.9.2). The Command Publisher and Command Responder provide for simple one-way message broadcasts (see Section 3.9.3).

JeroMF also includes a special type for Requester, called RegistrationClient, that registers services with the optional Registry process. See Section 3.9.4 for details about how this special communicator is automatically used for service registration.

All communicators can send and receive encrypted or unencrypted messages. For encrypted messages, a communicator may use either asymmetric encryption based on a public-private key pair or symmetric encryption based on a shared secret key. For asymmetric encryption where the Communicator needs to encrypt or decrypt with a private key, the ZmqService needs to give the Communicator the name of the key pair and the password for opening the private key. It should get these values from the Settings object. Also, this private key and its password should be kept at secure place and not shared. For asymmetric encryption where the Communicator needs to decrypt or encrypt a message with a public key, it can ask the containing ZmqService object to lookup the public key by name. If the distributed application is using a Registry, then a

ZmqService can use the Registry discover a public key if it is not already known.

The Communicator class contains a Boolean property, isActiveObject. IsActiveObject represents whether the communicator is an active or a passive object, depending upon isActiveObject is true or false respectively.

3.7.2 Communicators Running in Active Mode

Figure 3.12 illustrates run method of the Communicator class which is invoked only if communicator is active. An active communicator will run on its own separate thread. The run() method is invoked when communicator starts and is an application of the Template Method pattern, with the initialize(), doSomething(), and doSomeWaiting() methods representing those parts that may be specialized.

```
public void run() {
    try {
        initialize();
        communicatorStatus = RuntimeStatus.Running;
        while (keepGoing()) {
            boolean didSomething = doSomething();
            if (keepGoing() && !didSomething) {
                doSomeWaiting();}
        }
    }
    catch (CommunicatorException e) {
        stop();}
    if (communicatorStatus == RuntimeStatus.Running){
        stop();}
    cleanup();
    communicatorStatus = RuntimeStatus.Stopped;
}
```

Fig 3.12 Code snippet of run() method in the Communicator

The initialize() method creates socket, sets up the message handlers and initializes the statistics object. The doSomething() method furthers the work of communicator, i.e., sends or receives the message. It returns true, if the method there should be no waiting afterwards (a call to doSomeWaiting); otherwise false. The doSomeWaiting() method should gracefully wait until there is a something to send or receive.

3.7.3 Communicators running in Passive Mode

In the passive mode, a communicator does not have its own thread. Instead, its

service must explicitly call its send and receive methods, after starting the communicator.

Specifically, Communicator class supports both synchronous and asynchronous communications.

3.7.4 Communicator's Working

A Communicator class object handles incoming messages, by maintaining a HashMap with key as message class and value as instance of HandlerDefinition class. Handler Definition captures the definition of a function, which returns a reply message to the request message. Handler function is executed when a communicator receives a message and it sends function's return message as reply. Specifically, it generalizes the reply behavior of communicator. The handler definitions can be added or removed by calling addMessageHandler() and removeMessagehandler() methods.

Specialization of Communicator class will add message handlers for different type of messages it wants to respond. These message handlers can be added at runtime as well.

Moreover, Communicator maintains message statistics like number of messages sent, received and accepted by maintaining the instance of CommunicatorStats Class.

The start() and stop() methods should be called by user, specifically the service of which this communicator is a part. The start() method starts this communicator and stop() method gracefully stops it.

3.8 Messages

Since communicators send and receive messages, JeroMF provides a base class, called Message, for implementing message structures quickly.

Therefore, Message class is generalizations of all messages that will be sent or

received by the communicator. The Message class has some pre-defined specializations (See Figure 3.13) and will be specialized further by applications. For example, the UsedCar application specializes Message class for every type of message communicator wants to send or receive. ListCars, GetPriceForCar, ListCars and CarPrice are specializations of Message class in this sample application. Developers simply have to create specializations of this base class and then define appropriate data members with getters and setters.

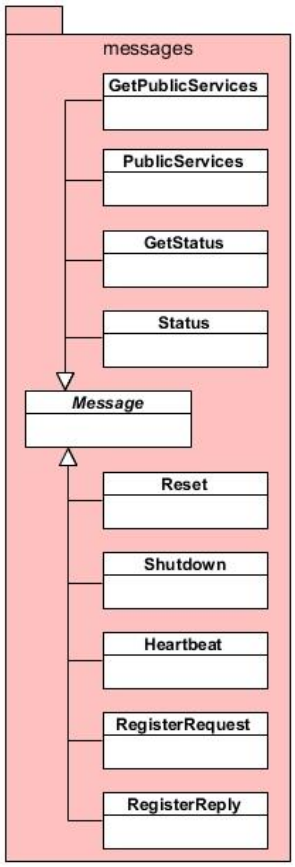

Figure 3.13 - Messages

3.9 Communication Patterns

With the existing specializations of Communicator class and the ability for developers to create their own specializations, JeroMF can supports a wide variety of communication patterns. The following subsections describe them in detail.

3. 9.1 Requester – Responder Communications

In this pattern, a Requester object initiates reliable request-reply conversation by calling the doRequestReply() method defined in the Requester class (see Figure 3.14).

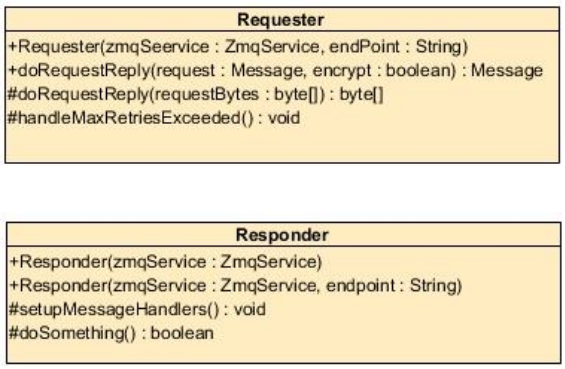

Fig-3.14 Requester-Responder

This method sends the provided message to the connected Responder and then waits for a response. If no response comes within the specified timeout, it tries again and continue to do so, until either it receives a response

message or the maximum retries has been reached.

A requester can be used either as an active or passive communicator, by setting isActiveObject property as true or false.

If it is being used as a passive communicator, then Requester class does not need to be specialized (but still can be) and the user of a Requester will call doRequestReply() to perform synchronous request-reply style communications.

If it is being used as an active object, the Requester class needs to be specialized and doSomething() method will be overridden to perform the desired work. The specialized doSomething() implementation may call doRequestReply() as needed. The doSomeWaiting() and handleMaxRetriesExceeded() method may also be overridden to implement custom behaviors.

Responder class includes the standard implementations of reply side of request-reply behavior. It can configure reply messages corresponding to incoming messages by adding message handlers.

The Specialization of Responder class only need to specify how to process one incoming request by overriding processIncomingMessage() method which is optional.

3.9.2 Passive Requester - Active Responder Communicator

The Passive Requester class object interacts with the Active Responder class object. Here Passive Requester acts like the requester but uses a "REPLY" type socket and binds to a port. Similarly, Active Responder acts like a responder but uses a "REQ" type socket and will connect to passive requester. This allows for multiple Active Responders that form a worker pool for handling requests from passive Requester (See

Figure 3.15). After Active Responder has connected to Passive Requester, Active Responder will send a "READY" message to the Passive Requester. This READY message means the Active Responder is available and ready to receive a request.

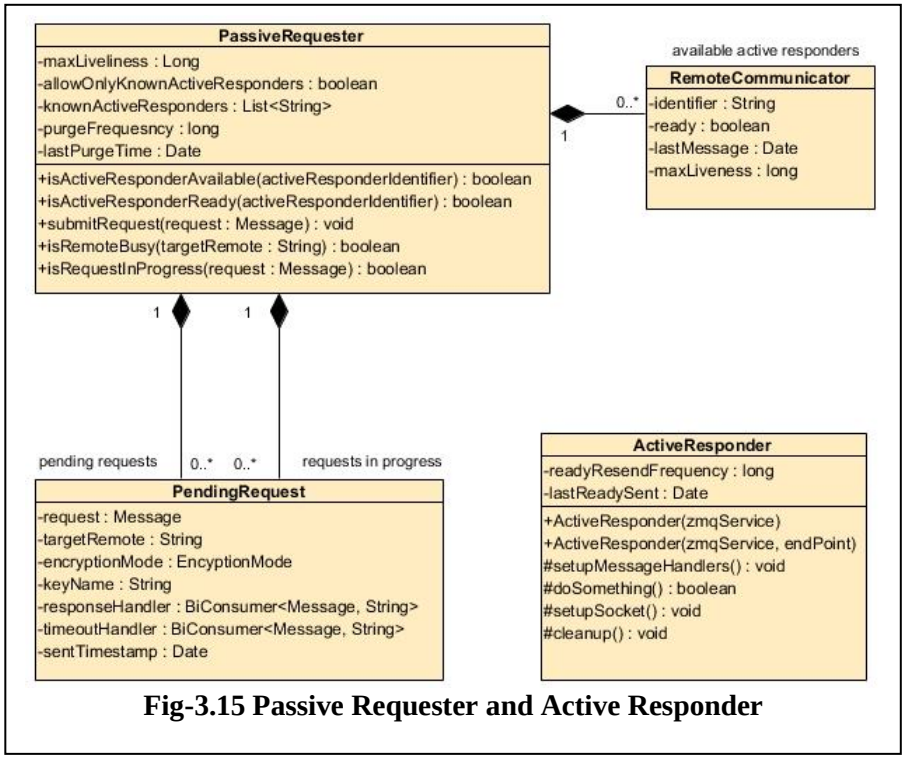

Fig-3.15 Passive Requester and Active Responder

In terms of the underlying request-reply pattern required by a "REP" type socket (on the Passive Requester side), the READY message acts like the request message. Eventually, the Passive Requester will send a request message to the Active Responder. Even though this message is a request message, it fulfills the role of a reply in that underlying pattern. After processing the request message, the Active Responder will send back a response message. Like the READY message, this response acts as request in underlying pattern and implies that the Active Responder is available and again ready of a new request.

A Passive Requester is an active communicator. When it tries to do work,

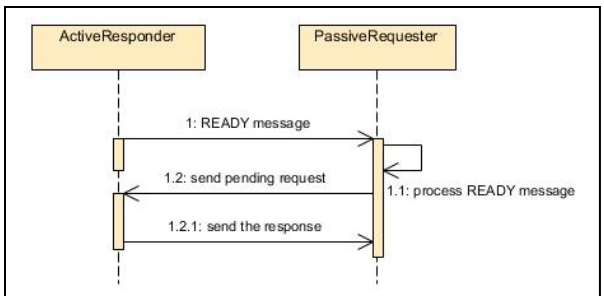

Fig-3.16 Sequence diagram of passive requester and active responder

it does the following (See Figure 3.16):

It tries to receive a message, which could be a READY or a response from an Active Responder. If received message is READY, then move Active Responder to list of available communicators. If it's a Response, then remove it from request in progress, and make it available again.

It will check if there are any previous requests for which no response has come back within the timeout limit. If so then, it will remove those requests as they are still in progress and remove those Active Responders from list of available ones.

And at last try sending out a pending request.

A client using a PassiveRequester should submit request via the submitRequest method. If desired, the client can specify a custom response handler and custom timeout handler. If a custom response handler was not specified, the client can setup message handlers for the expected message types.

An Active Responder on other side, being an active communicator continuously listens for requests and sends a reply. According to ready resend frequency, it sends READY message to passive requester, continuously so that it becomes an available active responder in timely manner. When active responder is stopped, it sends STOP message to Passive Requester, so that active responder is removed from the list.

Specialization of Active Responder class will setup message handlers to process the incoming requests.

3.9.3 Command Publisher - Command Responder Communicator

Command Publisher publishes system-wide commands on a PUB-SUB channel, and Command Responder subscribes and handles these commands. Following are

possible outgoing commands:

Shutdown Message is specialization of Message class. This is sent for stopping the process with all its services.

Reset Message class is specialization of Message class. This is sent for the process to register with Service Registry and thus get new Registration for the microservice.

CommandPublisher class contains a long property, remainingTimeToReset (See Figure 3.17). It represents the time after which Reset Message will be published. RemainingTimeToReset is initialized with resetInterval property encapsulated in settings of zmqService. When CommandResponder receives the Reset message it calls setRegistration method of zmqService and registration to null. See Section 3.9.4 for registration.

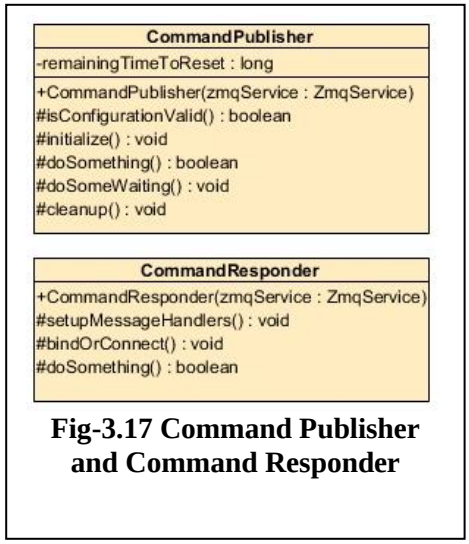

Fig-3.17 Command Publisher and Command Responder

Shutdown message is published when CommandPublisher's stop() method is called. When CommandResponder receives the Shutdown message it stops the containing microservice.

3.9.4 Service Registration Through a Registration Client

Registration Client is an active requester-type communicator that registers the containing zmqService with a special process, called the ServiceRegistry.

The Service Registry, which is included in JeroMF, is not part of SDK, but an

individual process that can be executed as part of a distributed application.

Every ZmqService can have its own private/public key-pair that it uses to register with the Service Registry or for other asymmetric encryption. If it needs symmetric encryption, it will obtain the shared symmetric key from the Service Registry.

The Service Registry was implemented using JeroMF, and specifically, a ZmqService with an ApiResponder and CommandPublisher. See Appendix B.2 for instructions on starting and stopping the Registry.

If developer wants to use the built-in Registry and registration process, then a ZmqService will use Registration Client, which is a Communicator to communicate with Registration Service (See Figure 3.18).

To receive Registration from the service registry, the RegitrationClient sends a Register Request message to the registry.

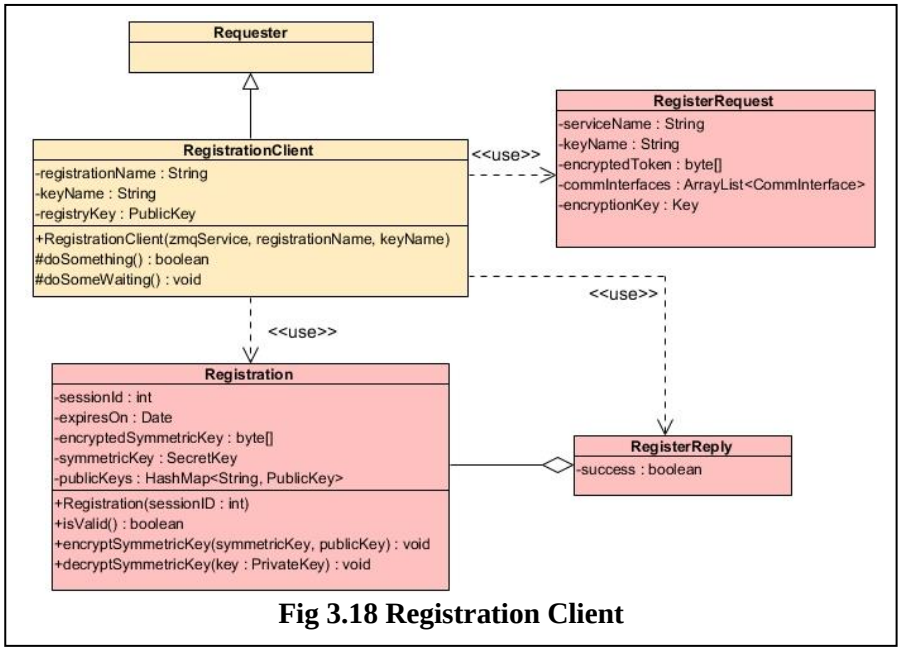

Fig 3.18 Registration Client

This request includes the name of containing service, by which the registration will be listed, the public key for the registry and the encrypted token.

The token is encrypted with the private key of the specific key pair. This helps the Service Registry to ensure that the Registration request came from an authorized client.

Specifically, if it can decrypt the token, then it knows that the sender had the correct private key.

Upon successful registration, the Registry will send back a Register Reply message, encrypted with its private key. The Registration Client will decrypt this reply with the Registry's public key.

Registration request is verified with the use of two different key pairs in registration conversation. The Service Registry can authenticate the client and the client can authenticate the Service Registry.

Register reply message includes Registration class object for the requesting service. Registration is a communication session. It contains a secret key, which itself is encrypted and date/time when this session will expire. Registration client will decrypt encrypted secret key with its private key. This Secret key will be used by the communicators of zmqService for subsequent communications with symmetric encryption.

After registration is setup with the service, this registration session will expire on date saved in expireOn property of Registration class. Then a new one will be setup through another registration conversation.

If the ZmqService has a RegistrationClient, then it will try to connect with the Registry and setup the Registration. The RegistrationClient will call the setRegistration method when it receives a registration. Any specialization of ZmqService or external object can get the current registration by calling getRegistration method.

3.10 Envelope

The **Envelope** class contains functionalities for converting Message class object to byte array and vice versa (See Figure 3.19) for unencrypted messaging.

The functionalities for asymmetric encryption/ decryption are in **EnvelopeWithAsymEncryption** and functionalities for symmetric encryption and decryption are in **EnvelopeWithSymEncryption**. These are specializations of Envelope class.

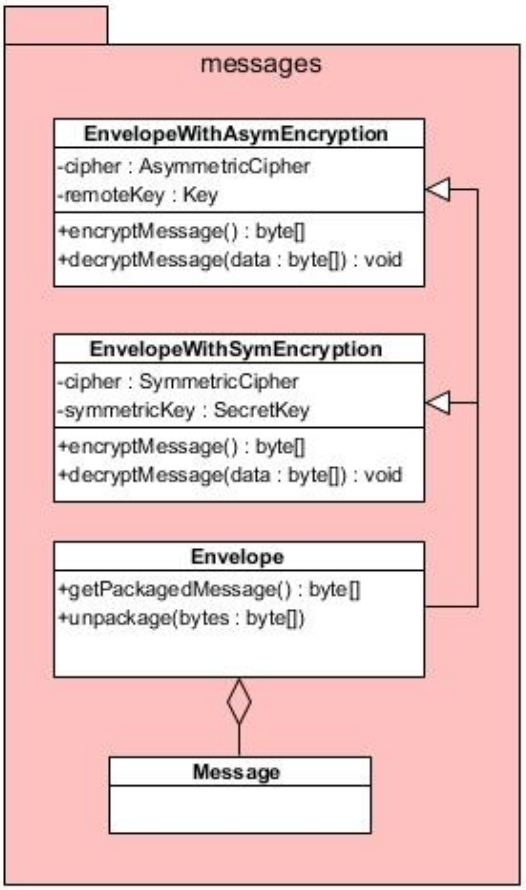

Fig 3.19 Envelope and Message

CHAPTER 4

TESTING

4.1 Introduction

Software testing is a broad term encompassing a wide spectrum of different activities, from the testing of a small piece of code by the developer (unit testing), to the customer validation of a large information system (acceptance testing), to the monitoring at run-time of a network-centric service-oriented application [13]. To ensure the quality and performance of JeroMF, unit testing, and integration testing is performed.

4.2 Unit Testing

Unit testing is recognized as an essential phase to ensure software quality, because by scrutinizing individual units in isolation it can early detect even subtle and deeply-hidden faults which would hardly be found in system testing [13].

In JeroMF thorough executable unit test cases were done for all the classes and for each service based on path and input validation testing techniques [14][15]. The thoroughness of the test cases for these two techniques was designed so that every possible branch of condition, boundary conditions for loops was tried such that collectively, the test case executes every line of code. Every possible exception was tried. At least one example of each element of partitioning of the input domain was represented. During concurrency, the time was considered part of the input domain such that various combinations of possible overlapping for critical sections are covered.

During the unit testing, we discovered that some of the declared exceptions from JeroMQ and other 3rd party libraries are impossible to stimulate in automated test cases. So, our coverage for unit testing is not 100%, but it is very close.

4.3 Integration and System Testing

Integration and System Testing is required to handle the complex spectrum of possible static and dynamic dependencies between classes [13].Integration test cases were written to ensure that component works appropriately together. Like the unit test cases, these will be implemented as executable methods in test classes.

However, each of these test cases must ensure that other services are running and, if not, start them up before executing and shut them down afterwards. To this end, we created some utility components for checking the status of another service, for launching a process that contains that service, and for eventually shutting that process down. These utilities components allow us to create automated integration and system test cases, giving us confidence that the individual components of JeroMF are working together correctly and that the framework is satisfying its requirements.

4.4 Validation

To see whether end goals were achieved or not, JeroMF requires using it to develop real distributed applications. Over the last 20 years, Utah State University has developed several distributed applications for the Utah Department of Health, including an information broker, called the *Child Health Advanced Record Management* (CHARM) system. This system allows health-care professionals to view a wide range of health-care data for a given child from multiple data sources, securely and in real-time. To do its job, CHARM must monitor and interact with multiple data sources and data consumers, matcher child records across the data sources, identify special situations about which

health-care professionals need to be alerted, and monitor itself.

This distributed application, which has been operating since 2006, seemed like a good candidate to re-design and re-implement using JeroMF. It is complex, requires high levels of security, maintainability, and extensibility. So, as an initial case study, we selected a major portion of this system, called the Sync Facility, and re-built this subsystem using JeroMF.

After refactoring into microservices, the Sync Facility ended up with 16 different types of services, hosted in 13 processes. The refactoring simplified the architectural design of the Sync Facility and improved its ability to be tested and deployed. Though antidotal evidence, the developers also believe that the new Sync Facility will be more maintainable and extensible.

CHAPTER 5

RELATED WORK

Many libraries and frameworks have been developed to reduce the complexity of creating distributed applications. For discussion purposes, I have organized the related work into three areas. The following subsection discusses each of these areas.

5.1 Service-oriented distributed computing

Service-oriented architecture aims to achieve loose coupling between services which are fundamental units of distributed application. One example of a framework in this area is the Globus Took Kit, which supports development of service-oriented distributed computing application [16]. It provides components that solve common problems when building distributed services and applications. Specifically, it provides services for resource management, execution management, data access and movement, replica management, service discovery and many more with standard-based security infrastructure for authentication and authorization. It uses SOAP as the messaging protocol and encryption for security [16].

5.2 Master/worker Architecture and event loop

DistGear is an open source software built on top of communication to provide developers with a programming abstraction to easily handle distributed tasks and control machine coordination in tasks [17]. This programming framework is based on event driven model. It is implemented in Python and based on coroutines and ZeroMQ [17].

5.3 Other Frameworks for microservices

WSO2 Microservices Framework for Java is a lightweight, fast runtime

annotation-based programming model with container-based deployment for developing and running micro services. It provides integrated security of microservices via token validation. It provides high scalability and reliability with HTTP/HTTPs transports based on Netty 4.0, with full control of streaming by developer [18].

Frameworks like SwaggerHub, Jersey and Spring Boot is widely used in industry as Microservice Framework in JAVA.

EventuateTram and ES are micro service frameworks for building asynchronous microservices for solving Distributed data management problems. It provides easy implementation of eventually consistent business transactions that span multiple microservices. Also, provides faster and more scalable querying using CQRS (Command Query Responsibility Segregation) views [19].

Axon is open source framework that solves application complexity in event-driven microservices. It is based on architectural principles like domain driven design and CQRS. It helps to create scalable and extensible applications while maintaining application consistency [20].

CHAPTER 6

SUMMARY

Our initial experience with JeroMF has provided preliminary evidence that it is valuable framework for implementing distributed applications based on microservices and JeroMQ. Its BaseProcess class makes it easy to define new service containers that can run on bare-bones Java platforms, i.e., a platform with no web servers or application servers. Its BaseService and ZmqService classes make it easy to create custom microservices that can implement diverse and sophisticated functionality. The predefined Communicator and Message classes allow developers to implement common styles of communication and provide excellent starting points for implementing application-specific communication protocols. Also, the Communicator class makes it easy for developers to use either asymmetric or symmetric encryption. Furthermore, the optional Registry process can act like a key store for the public keys of registered services, simplifying key management.

The JeroMF services also have built in monitoring logic that can allow monitoring processes to either actively query the service status or receive periodic updates from services. Services can also track statistics about workloads and message traffic, and then provide that information to monitoring processes for analysis. Finally, the standard Command Responder for a service provide a simple but secure way to shut down or restart services.

Despite its rich set of features, JeroMF is still in its infancy. Several important enhancements to JeroMF could be made in the near future. First, creating other specializations of BaseService, like ZmqService, that would support different messaging

libraries. For example, an HttpService that uses HTTP [21] instead of JeroMQ and that has built-in support for RESTful [22] operations. Second, implementing and testing extensible services that will act as request proxies and load balancers.

The URLs for JeroMF's public repositories are as follows:

https://bitbucket.org/usucssedevelopment/base.git

https://bitbucket.org/usucssedevelopment/registry.git

https://bitbucket.org/usucssedevelopment/utils.git

https://bitbucket.org/usucssedevelopment/jeromf-examples-usedcars.git

REFERENCES

[1] J. Lewis and M. Fowler, "Microservices," *martinfowler.com*, 25-Mar-2014. .

[2] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study.," presented at the CLOSER, 2018, pp. 221–232.

[3] D. Kerr, "The Death of Microservice Madness in 2018," *Dave Kerr's Blog*, 12-Jan-2018. .

[4] P. Hauer, "Microservices in a Nutshell. Pros and Cons.," *Phillip Hauer's Blog*. .

[5] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, no. 3, pp. 301–310, Jul. 2017.

[6] "ZeroMQ." [Online]. Available: https://en.wikipedia.org/wiki/ZeroMQ.

[7] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[8] M. Sústrik, "ZeroMQ," *Introduction Amy Brown and Greg Wilson*, 2015.

[9] D. Spinellis, "Git," *IEEE Software*, vol. 29, no. 3, pp. 100–101, Jun. 2012.

[10] "ZeroMQ: Solving the Challenges of Distributed Systems." [Online]. Available: https://openit.com/zeromq-solving-the-challenges-of-distributed-systems/.

[11] A. S. Evans, "Reasoning with UML class diagrams," in *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998, pp. 102–113.

[12] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[13] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams."

[14] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, May 1978.

[15] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering*, no. 3, pp. 247–257, 1980.

[16] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," *Journal of Computer Science and Technology*, vol. 21, no. 4, p. 513, Jul. 2006.

[17] J. Ma, B. An, X. Chen, and D. Cao, "DistGear: A Lightweight Event-Driven Framework for Developing Distributed Applications," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 2018, vol. 01, pp. 106–115.

[18] "WSO2 Microservices Framework for Java." [Online]. Available: https://wso2.com/products/microservices-framework-for-java/.

[19] C. Richardson, "Solving distributed data management problems in a microservice architecture." [Online]. Available: http://eventuate.io/.

[20] "Axon Framework: Open source framework for event-driven microservices and domain-driven design." [Online]. Available: https://axoniq.io/product-overview/axon-framework.

[21] J. F. Reschke and R. T. Fielding, "Hypertext Transfer Protocol (HTTP/1.1): Authentication." [Online]. Available: https://tools.ietf.org/html/rfc7235. [Accessed: 14-Aug-2019].

[22] "What is RESTful API? - Definition from WhatIs.com," *SearchMicroservices*. [Online]. Available: https://searchmicroservices.techtarget.com/definition/RESTful-API. [Accessed: 14-Aug-2019].

APPENDIX A

SETTINGS FILE

An example of Settings File is shown below

```
priv-keys-location=/TestPrivateKeys/
pub-keys-location=/TestPublicKeys/
key-pair-name=test1
priv-key-password=test
public-command-ep=tcp://10.11.12.13:1234
registration-ep=tcp://10.11.12.13:2345
api-ep=tcp://*:2222
public-api-ep=tcp://10.11.12.13:2222
main-sleep=1000
timeout=500
max-retries=4
db-driver=test.postgresql.Driver
db-connection=jdbc:postgresql://1.2.3.4:5432/charm
db-username=abc
db-password=test123
max-ready-aliveness=1000
ready-resend-frequency=500
purge-frequency=60000
reset-interval=1000
heartbeat-interval=1000
stats-window-size=24
stats-slot-time-duration=1000
```

APPENDIX B.1

JEROMF CODE

The URLs for JeroMF's public repository is as follows:

https://bitbucket.org/usucssedevelopment/base.git

Below is the Read me file.

```
# README #

This repository will hold the base process, message, and worker
components for all of CHARM.

### How do I get set up? ###

* Summary of set up
* Configuration
* Dependencies
* Database configuration
* How to run tests
* Deployment instructions

### Contribution guidelines ###

* Make sure that there are executable test cases that provide good
coverage
* Do a code review with Stephen Clyde or Aihua Tong
* [Learn Markdown](https://bitbucket.org/tutorials/markdowndemo)

### Who do I talk to? ###

* Stephen Clyde, Utah State University, Stephen.Clyde@usu.edu, 435-
764-1596
```

APPENDIX B.2

Service Registry

The Services registers to Service registry to get symmetric keys for encryption.

The Service Registry uses javax.crypto.KeyGenerator with AES to generate symmetric key and Bouncy Castle as encryption provider.

The URLs for JeroMF Registry's public repositories are as follows:

https://bitbucket.org/usucssedevelopment/registry.git

Below is the Read Me File

```
# README #

This repository is for the Registry component of CHARM which
includes the command server and the registry server.

### How do I get set up? ###

* To run, you specify the private key's password in a commandline
parameter

### Contribution guidelines ###

* Writing tests
* Code review
* [Learn Markdown](https://bitbucket.org/tutorials/markdowndemo)

### Who do I talk to? ###

* Stephen Clyde
```

APPENDIX C

USED CAR SETTINGS

The URLs for JeroMF Registry's public repositories are as follows:

https://bitbucket.org/usucssedevelopment/jeromf-examples-usedcars.git