FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

A Framework for Distributed Systems Based on The Actor Programming Model and Dart language, Which Unifies Applications Across Devices, Clients and Servers, and Supports Features for Hot Deployment and Migration of Actors

Sushil Man Shilpakar

FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

A Framework for Distributed Systems Based on The Actor Programming Model and Dart language, Which Unifies Applications Across Devices, Clients and Servers, and Supports Features for Hot Deployment and Migration of Actors

Ein Framework für verteilte Systeme auf der Basis des Actor-Programmiermodells und der Dart-Programmiersprache, die Anwendung in Endgeräten, Clients und Servern erlaubt und die Möglichkeit für Hot Deployment und Migration der Actors bietet

Author:	Sushil Man Shilpakar
Supervisor:	Prof. Hans Arno Jacobsen
Advisor:	Richard Billeci
Submission Date:	TODO: Submission date

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, TODO: Submission date Sushil Man Shilpakar

Acknowledgments

TODO

Abstract

TODO

Contents

1 Introduction

1.1 Background

TODO

1.2 Goals

TODO

2 Literature Review

2.1 Message Passing Paradigm

Message passing is sending a message from one process, component or actor to another. The recipient of the message chooses further processing based on the pattern or content of the message. Message passing is the loosest type of coupling. The components of a system are not dependent on each other; instead they use a public interface to exchange parameterless messages[Cel11]. Hence, systems that implement message passing paradigm are easily scalable and efficient. Such systems are easy to replicate and easy to make fault-tolerant. [Arm10].

The systems which communicate solely by message passing do not have a shared state. Such systems can be easily divided into isolated components. This makes the overall architecture of a system easy to understand and simplifies the separation of problems within the system [Arm10].

Message passing can either be synchronous or asynchronous. Synchronous message passing is the communication between two components where both the sender and the receiver of a message are ready to communicate. As the sender of a message has to wait for a response from the receiver, the sender is usually blocked until it gets a reply. Whereas, in asynchronous message passing, the sender simply sends a message and continues to do other tasks. The receiver does not have to be ready to accept the message when the sender sends it [Agh85]. Thus, an asynchronous message passing is non-blocking in nature.

2.2 Actor Programming Model

Actor programming model is a programming paradigm designed for concurrent computation. The concept of an actor was originally introduced by Carl Hewitt[AH85]. He, along with Agha[Agh85] have been involved in development of the actor theory as well as its implementation.

An actor is a fundamental unit of computation. It is neither an operating system process nor a thread, but a lightweight process. It embodies three essential things:

- Processing

- Storage
- Communication

Actors have addresses and there is a many-to-many relationship between actors and addresses.

Actors communicate with each other in a non-blocking way by asynchronous message passing, which removes the need of explicit locks. An actor can send message to actors in same system or another system. An actor can also send message to itself, which is how a recursion is achieved. An actor can send a message to target actor only if it has the address of target actor. Agha ([Agh85], p35) lists three ways in which an actor, upon accepting a message, can know the address of the target actor:

- the target was known to the actor a before it accepted the message
- the target became known when the message was accepted because it was contained in the message, or,
- the target is the new actor created as a result of accepting the message

To buffer incoming messages, each actor has a mailbox. A mailbox is a queue of messages that have been sent by other actors or processes and not yet consumed, where mailbox is also an actor. According to Hewitt, the order in which the messages are delivered is non-deterministic [Nin].

After receiving a message, an actor may perform following actions. [AH85]

- Create other actors
- Send messages to itself, other known actors or reply to the actor who sent the message
- Designate how it is going to handle next message, i.e. Specify a replacement behavior

Actors do not have a shared mutable state. All mutable state is private to the actor and all shared state is immutable. Actors communicate with each other by asynchronous message passing which is also immutable. Each actor processes only one message at a time, and unless it is a broadcast message, a message is not processed multiple times.

An actor exists in a system. An actor system is a group of actors working together in certain hierarchy.

In the actor model, concurrency is inherent because of the way it is designed. Also, there is no guarantee that the message sent to an actor will arrive sequentially [Nin]. Nevertheless, Akka framework [section 2.5] guarantees the order of delivery of messages between two actors, provided that there are no intermediary actors in between[subsection 2.5.3].

Several programming languages like Act 1, 2 and 3, Acttalk etc. were created when actor system was newly introduced by Hewitt and Agha [Agh85; AH85].

2.2.1 Error Handling in Actor Model

Exception handling in actor model is based on idea of embracing the failures. The idea of embracing failures is also known as "let it crash" paradigm [subsection 2.3.1]. As the actors do not have shared state, this allows individual actor to fail without causing disruptions in the system. Since, an actor in an actor system is typically organized in a hierarchical structure, the actor which created a child actor can be used for supervising it. The idea of supervision in a hierarchical actor system helps to make the actor system fault tolerant [Erb12]. When an actor throws an exception the supervisor can respond to the exception in different ways. In Akka [section 2.5], the supervising actor usually reacts by either simply ignoring the exception and letting the actor continue, by restarting the actor or by escalating the error to its supervisor.

The "let it crash" style of programming is a non-defensive programming, which is implemented successfully in Erlang [section 2.3].

2.2.2 Differences from thread-based programming model

Thread-based Concurrency

In thread-based programming languages, the control flow of a program is divided into several threads for concurrency. The threads operate simultaneously and the control can switch from one thread to another non-deterministically. When two or more threads have a shared memory, concurrent modifications and accesses of the data might result in undesired behavior of the system, known as 'race condition'. To prevent this type of situations, such programming languages have locks. The locks let only a single thread at a time to run sequentially for the section of a program code [Sofa].

Generally, thread-based programming models are easy to understand and implement. But, the resulting program behavior is difficult to understand because of implicit context switches and release of locks, which may even lead to a deadlock situation [Sofa].

Actors based Concurrency

Concurrency in actor based programming languages are inherent because of asynchronous message-passing, pipelining, and the dynamic creation of actors. The concurrency in actors through pipelining is only constrained by the logical limits and the available hardware resources [Agh85]. The actors may carry out their activities in parallel as each actor resides in a completely separated space from the other actors, they are connected only via messages.

The actor based programming liberates the programmer from going into coding details about the parallelism and threads [Agh85].

As actors do not have shared state, thus the 'race condition' as discussed in section 2.2.2, do not arise in actor based programming.

2.2.3 Scaling up actor based systems

As actor based systems are highly concurrent [section 2.2.2], it is easy to scale actor based systems from a single core system to several of them across multiple data centers around the globe. The ideal case would be to just let a new system join the actor system and let it run some more actors during runtime, without the need of taking down the running system. The actors themselves do not need to know the physical location of other actors as they simply exchange messages based on the logical addresses. This makes them easy to scale up and out.

2.3 Erlang

Erlang is the first popular programming language that was based on the actor model[Vin07]. It was developed by Joe Armstrong in 1986 at the Ericsson Computer Science Laboratory and was made open-source in 1998. It was chiefly used in telephony applications as it was built to solve the problems of availability as well as scalability that existed in such applications. [Arm07]

Erlang 'processes,' which are essentially user-space threads rather than Unix processes or kernel threads, communicate only via message passing.

Erlang was designed for the writing applications that require high availability [Arm07]. It uses concurrent processes, which have no shared memory and communicate only via asynchronous message passing. This idea is similar to the actors model proposed by Agha and Hewitt [section 2.2]. Thus, programs written in Erlang are concurrent, distributed, fault tolerant and thread safe. The concurrency is built into the language itself, not the the operating system [Arm10].

When an application developed in Erlang is deployed in a multicore computer, it automatically takes advantage of those multiple cores. The Erlang processes spread over the cores. So, the programmers do not have to worry about threads [Arm10]. In Erlang, new changes in an application can be added to the system without taking it offline. This improves the availability of whole system. Thus, it simplifies the construction of software for implementing non-stop systems [Arm07].

Error handling in Erlang is different from most of the other programming languages. The error handling is based on a "let it crash" philosophy subsection 2.3.1 which is a non-defensive style of programming [Arm07; Arm10].

2.3.1 "Let it Crash" Philosophy

The core idea in "Let it Crash" philosophy is to let the failing processes crash and make other processes, which observe this process, detect the crashes and fix them [Arm10]. This idea is in sharp contrast to other programming languages, where programmers implement exception handlers and prevent a process from getting terminated.

The concept of "let it crash" leads to clear and compact coding [Arm10].

2.4 Scala Actors

Scala is a statically typed programming language which integrates functional as well as object-oriented programming [OR14]. Scala has its own library for actor programming.

In Scala, templates for actors with user-defined behavior are normal class definitions which extend the predefined Actor class.

The scala actors library provide concurrent programming model based on actors [section 2.2]. The actors in scala are fully inter-operable with the ordinary virtual machine threads [HO09]. Scala actors are lightweight and support around 240 times more actors to run simultaneously compared to virtual machine threads [HO09].

Both synchronous and asynchronous message passing can be used to in scala actors. The synchronous message passing is implemented by exchanging several asynchronous messages. The actors in scala can also communicate using 'futures'. When a future is used the requests are handled asynchronously and the sender immediately gets a representation of the future which allows sender to wait for the reply. [Hal]

The Scala Actors library is now deprecated and will be removed in future Scala releases[1]. The deprecation is in favor of the use of Akka actors [section 2.5].

2.5 Akka Toolkit

Akka is a toolkit and runtime for building highly concurrent, distributed and resilient message-drive applications on the JVM. [Incb] Akka uses lightweight actors for concurrency. The actors in Akka are based on Hewitt and Agha's model [Agh85; AH85] of actor programming. Akka provides a well defined API for developers to create large

[1] Scala actors are deprecated in version 2.10 [HT]

concurrent systems, which allows for easy scaling out. Akka is available for Scala as well as Java.

2.5.1 Actor Model in Akka

In Akka, actors are objects which encapsulate state and behavior. Similar to Hewitt and Agha's actor model [section 2.2], the akka actors communicate only by messages which are placed into the recipient's mailbox. Akka actors are the stringent form of object-oriented programming [Inca].

The 'ActorRef' in akka, represents an actor. It holds a reference to an actor. Its purpose is to facilitate the message sending to the actor it refers. An actor can also refer to itself using *self*. [Incc]

The Listing 2.1 [2], is a simple example of how actor programming in Akka is realized. In this example, the 'GreetingActor' which extends 'UntypedActor' must implement the *onReceive()* method. The *onReceive()* method is invoked for each message received by this actor. After receiving a message, the actor performs pattern matching to decide which code to execute. As in this example, if the message is a type of 'Greeting' object, it is concatenated with the string "Hello" otherwise, it is simply ignored.

Listing 2.1: A simple example of actor programming in akka [Incb]

```
public class Greeting implements Serializable {
  public final String who;
  public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  public void onReceive(Object message) throws Exception {
    if (message instanceof Greeting)
      log.info("Hello " + ((Greeting) message).who);
    }
}

ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(Props.create(GreetingActor.class), "greeter");
greeter.tell(new Greeting("Charlie Parker"), ActorRef.noSender());
```

[2]Source: T. Inc. *Akka Toolkit*. `http://akka.io`. Last Accessed: 2014-11-19

2.5.2 Actor System

An actor system in Akka is like an organization of actors where actors are arranged in a hierarchical structure. An actor, which performs certain task usually split up the task into smaller pieces. For this, it starts up other actors as its child actors, which it supervises subsection 2.5.5. As the actor which creates a new actor is the supervisor of the new actor, it is implicit that there can be only one supervisor of an actor.

By splitting up the tasks, the tasks become clear and structured. Furthermore, the resulting actors also becomes simplified and specialized in terms of which messages it should process and how it should react normally. It also becomes easy to handle the failures. If an actor cannot cope with a failure, it is escalated up in the hierarchy to its supervisor which is recursive unless it reaches up to the actor which can handle the failure. [Incc]

An actor instance in akka takes up roughly 300 bytes of memory, because of which it is possible to spawn millions of them in one actor system. As an actor system can grow very large and distributed, the order of message processing is not guaranteed. [Incc]

An example of the arrangement of actors in hierarchical structure is shown in Figure 2.1. The '\' is known as the "root guardian" and *user* is known as the "user guradian". Any actor created by a user falls under the user guardian. For instance, in the Figure 2.1, *actorA* and *actorB* are actors created by user, hence their supervisor is *user*. Again, the actors *actor1* and *actor2* are the child actors of *actorB*. The address of an actor in akka is arranged like the filesystem hierarchy. Hence, the supervision hierarchy as well as the path leading to the actor is comprehensible from its address.

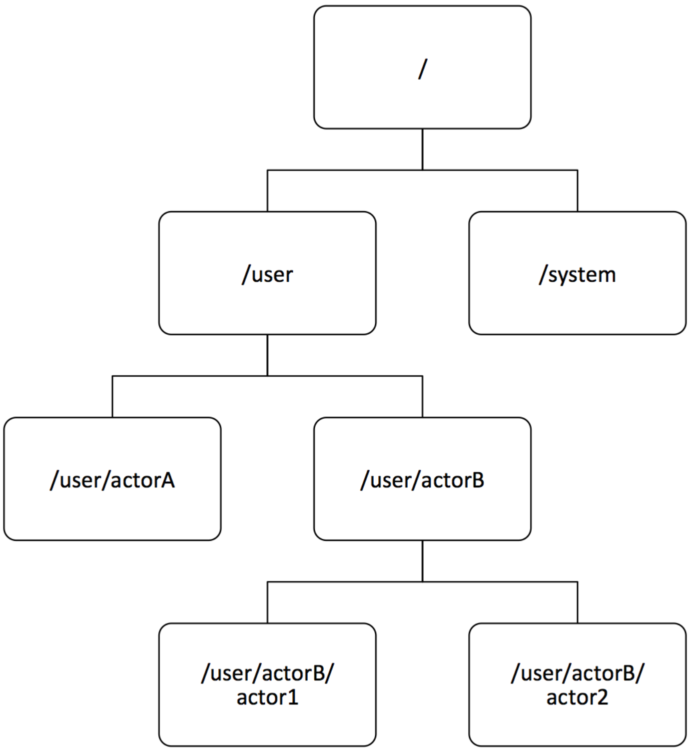

Figure 2.1: Hierarchy in actor system [Incc]

2.5.3 Message Passing in Akka

In Akka, every actor has an event driven message inbox known as the 'mailbox'. The mailbox buffers all the incoming messages until they are processed. When a message is sent from one actor to the other, the reference to the sender (an ActorRef), is automatically added to the message by default. Thus, during the message processing the recipient actor has a reference to the sender actor through the *sender* method. [Incc]

Akka guarantees the order of direct message delivery between two actors. It is only applicable when the mailbox implementation in receiver is FIFO mailbox and the communication takes place only between the two actors without the involvement of intermediary actors. For instance, if an actor A1 sends messages M1, M2 and M3 to actor A2. The message M1 will be delivered before M2 and the message M2 before M3. Meanwhile, if another actor A3 sends messages M4, M5 and M6 to A2 at the same time, the order of message delivery of M4, M5 and M6 will also be sequential. Nevertheless, there is no guarantee that the messages sent by A1 is delivered before the messages from A3; even if all of the messages from A1 were sent before A3 started sending messages to A2. Provided that the message sending from A1 and A3 are independent of each other. [Incc]

The Akka's documentation for Java, the rules mentioned for message sending are: [Incc]

- at-most-once delivery, i.e. no guaranteed delivery
- message ordering per sender–receiver pair

Here, 'at-most-once delivery' means that a message is either not delivered at all (i.e. lost) or delivered only once. Thus, there is no duplicate delivery of a message.

2.5.4 Shared mutable state

Since Akka runs on top of the Java Virtual Machine(JVM), there are some pitfalls that the programmer should be aware enough to avoid, which akka cannot enforce. The messages used to communicate between the actors should be immutable. If it is made mutable and the reference of it is sent by the sender to an actor which resides on same JVM, bugs such as race conditions and even some incomprehensible bugs might appear. Also, the sender method must not be closed over if the block of code could be run in another thread; for instance, sending (replying) a message to the sender actor inside a 'Future' block. In such cases, the sender reference must be captured into a local variable first. As the reference and behavior of sender may change over time [Incc].

2.5.5 Actor Supervision and Monitoring

The supervising actor in akka delegates tasks to its subordinates. It is also responsible for monitoring them for failures. When a child actor detects a failure, it suspends itself as well as its child actors and sends a message to its supervisor about the failure. According to Akka Java documentation[Incc], upon receiving the failure, the supervisor may opt to perform one of the four choices:

1. Resume the subordinate, keeping its accumulated internal state
2. Restart the subordinate, clearing out its accumulated internal state
3. Stop the subordinate permanently
4. Escalate the failure, thereby failing itself

When an actor is restarted by the supervising actor, the message that the actor was processing during the time of failure is lost and is not processed again. However, the messages that were in the mailbox of the actor remains safe and new actor resumes processing the other messages in the mailbox [Incc].

2.5.6 Routers

Routers are specialized actors that act as a load balancer for the actors that it supervises. Akka provides several built-in routing techniques: [Incc]

- Round Robin Router
- Random Router
- Smallest Mailbox Router
- Broadcast Router
- Scatter Gather First Completed Router
- Tail Chopping Router
- Consistent Hashing Router

It is also possible to use custom implementation of a router by extending *RoutingLogic* class from Akka's routing library.

2.5.7 Remote Actors in Akka

Actors in akka are location transparent; they reside in a logical hierarchy of an actor system through which the physical location of an actor in the network can be determined. The location transparency allows the akka applications to be developed locally but deployed into distributed systems simply by changing configurations. [Incc]

Actor systems in akka communicate in a peer-to-peer fashion and it is the foundation for Akka Clustering. Akka's Documentation for Java [Incc] lists the two design decisions for remoting:

1. Communication between involved systems is symmetric: if system A can connect to system B then system B must also be able to connect to system A independently.
2. The role of the communicating systems are symmetric in regards to connection patterns: there is no system that only accepts connections, and there is no system that only initiates connections.

The Listing 2.2 is an example taken from website of Akka[3] which is an example fo configuration and code for deploying actors in remote nodes.

Listing 2.2: A sample Akka Configuration and Code for Remote Actors [Incb]

```
// -----------------------------
// config on all machines
akka {
  actor {
    provider = akka.remote.RemoteActorRefProvider
    deployment {
      /greeter {
        remote = akka.tcp://MySystem@machine1:2552
      }
    }
  }
}

// -----------------------------
// define the greeting actor and the greeting message
public class Greeting implements Serializable {
  public final String who;
  public Greeting(String who) { this.who = who; }
}

public class GreetingActor extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  public void onReceive(Object message) throws Exception {
    if (message instanceof Greeting)
      log.info("Hello " + ((Greeting) message).who);
  }
}

// -----------------------------
// on machine 1: empty system, target for deployment from machine 2
ActorSystem system = ActorSystem.create("MySystem");

// -----------------------------
// on machine 2: Remote Deployment - deploying on machine1
ActorSystem system = ActorSystem.create("MySystem");
ActorRef greeter = system.actorOf(Props.create(GreetingActor.class), "greeter");

// -----------------------------
// on machine 3: Remote Lookup (logical home of "greeter" is machine2, remote
    deployment is transparent)
```

[3] T. Inc. *Akka Toolkit*. http://akka.io. Last Accessed: 2014-11-19

```
ActorSystem system = ActorSystem.create("MySystem");
ActorSelection greeter =
    system.actorSelection("akka.tcp://MySystem@machine2:2552/user/greeter");
greeter.tell(new Greeting("Sonny Rollins"), ActorRef.noSender());
```

2.5.8 Clustering in Akka

Akka does not have a central server. Instead, it uses peer-to-peer based gossip protocol to form a cluster. Thus, there is no single point of failure or single point of bottleneck in the system. But, adding a new member in a distributed system requires significant amount of time because of the nature of the gossip protocol. In a benchmark [Nor] performed by Patrik Nordwall[4], adding nodes to a cluster of at least 1500 nodes took around 15 to 20 seconds in average.

2.6 The Dart Language

2.6.1 Overview

Dart is an open-source, class-based, single-inheritance, pure object-oriented programming language developed by Google [ECM14]. Dart language is inspired by Smalltalk, Strongtalk, Erlang, C# and JavaScript [Lad].

Dart codes are not compiled before running; the Dart Virtual Machine(VM) reads and executes the source code. Dart provides a homogeneous system that encompass both client as well as server as the Dart VM can be embedded in browsers. A version of Chromium – 'Dartium' already has Dart VM built into it [Lad].

Programs in Dart are optionally typed. They can be executed in two modes checked mode and production mode. In checked mode incorrect static type annotations produce compile time errors. Whereas, in production mode, type annotations are completely ignored [ECM14].

Furthermore, Dart allows developers to code in a uniform way for both server as well as client since Dart Virtual Machine (VM) can be embedded in browsers. A variant of Chromium browser — Dartium browser has an embedded Dart VM.

Dart has automatic garbage collecting system, which means the memory occupied by objects which are not in use and which do not have any reference are reclaimed periodically.

Developers of dart believe that JavaScript has been pushed to its limit and the web apps developed in JavaScript are far too slow even though JavaScript engines are

[4]Patrick Nordwall is a developer of Akka at Typesafe Inc.

fast. They claim Dart offers a better solution to build larger and more complex web apps[Kat12].

Some of its important features are:

- Easy to learn syntax
- Compiles (Translates) to JavaScript
- Runs in client as well as on server
- Dart supports types, but it is optional
- Can scale from small script to large and complex applications
- Support safe concurrency with isolates
- Support of code sharing

2.6.2 Advantages of Dart

- Translates to JavaScript so that the code can be run in the web-browsers that do not have Dart VM yet
- Currently in 20th position in most popular programming languages
- Optionally typed language

2.6.3 Dart and JavaScript

Codes written in Dart can be translated to JavaScript using a tool – ‘dart2js’. The ‘dart2js’ is bundled with the Dart SDK (Software Development Kit). As the popular web browsers like Mozilla Firefox, Google Chrome, Safari do not have Dart Virtual Machine embedded, the ability to translate source code from Dart to JavaScript lets the dart programs run in any modern browser without needing to manually port the source code to JavaScript.

2.6.4 Asynchronous Programming in Dart

Most programming languages use callback functions for asynchronous programming. Dart provides some additional alternatives along with callback functions – Future and Stream objects. A ‘Future’ is a promise for a result which will be returned after an arbitrary amount of time. A ‘Stream’ is a way to get a sequence of values, such as events, data from ports etc.

2.6.5 Isolates

Although Dart programs are single threaded, concurrency is supported via actor-like entities called isolates. An isolate has its own memory and own thread of control. Message passing is the sole way to communicate between isolates. No state is ever shared between isolates. Isolates are created by spawning [ECM14].

An isolate has its own heap memory different from the main isolate (the top level isolate). It is possible for the child isolate to throw exceptions and errors such as by exhausting its memory. If the exceptions are not handled properly, it forces the isolate to be shutdown.

In Dart, when an isolate is spawned, usually the initial message contains a sending port, so that spawner and "spawnee" can communicate with each other. The "spawnee" can later use the same port to reply to the spawner. An isolate can spawn another isolate which can further spawn other isolates and have control over them. Thus, the spawner can supervise the "spawnee". The spawner can pause the "spawnee" or terminate it [Goo] [5].

Modern web browsers, even on mobile platforms, run on multi-core CPUs. To take advantage of all those cores, developers traditionally use shared-memory threads running concurrently. However, shared-state concurrency is error prone and can lead to complicated code. Thus, instead of threads, all Dart code runs inside of isolates. Each isolate has its own memory heap, ensuring that no isolate's state is accessible from any other isolate [Kat12].

Spawning an Isolate

There are two ways to spawn an isolate: using *Isolate.spawnUri()* or using *Isolate.spawn()*. The *Isolate.spawn()* uses top level function to spawn an isolate. The top level function may reside in the same class or may belong to another class. The *Isolate.spawnUri()* spawns an isolate using the source code of a file from a given location. The location can be a remote http/https URI or a path to source file in local disk. To spawn an isolate using *Isolate.spawnUri()*, the source file must have an entry point function *main()*. The newly spawned isolate shares the same code as the spawner isolate [Goo].

Communication Between Two Isolates

After an isolate is spawned, it is recommended to send its SendPort to the spawner isolate so that the "spawner" and "spawnee" can communicate via message passing. SendPort and ReceivePort are used by the isolates to communicate with each other.

[5] Pausing and terminating an isolate is not available in Dart 1.7.2 or older versions

The sender isolate uses the SendPort of the ReceivePort of the target isolate to send a message.

The example in Listing 2.3 shows how an isolate is spawned using *spawnUri()* and how to perform basic communications between two isolates in Dart. As shown in this example, the message sending is performed after the SendPorts are exchanged between the isolates.

Listing 2.3: A simple example of isolate communication in dart

```
//sample.dart
import 'dart:isolate';

main(var args, SendPort sendPort) {
  ReceivePort receivePort = new ReceivePort();
  SendPort sendport;
  sendPort.send(receivePort.sendPort);

  receivePort.listen((var message) {
    if(message is SendPort) {
      sendPort = message;
    } else if(message is String) {
      print("Received: $message");
    } else {
      sendPort.send("Unknown Message");
    }
  });
}

//app.dart
import 'dart:isolate';

main() {
  ReceivePort receivePort = new ReceivePort();
  SendPort sendPort;
      Isolate.spawnUri(Uri.parse("sample.dart"),null,receivePort.sendPort); //
      Spawns an isolate from sample.dart file

  receivePort.listen((var message) {
    if(message is SendPort){
      sendPort = message;
      sendPort.send(receivePort.sendPort);
      sendPort.send("Hello");
      sendPort.send(["a", "list", "datatype"]);
    } else if(message is String) {
      print("Reply: $message");
```

```
    }
  });
}
```

Difference from Actor

Although, Dart isolates do not have shared state and use message-passing as the only means of communication between two isolates, the isolates differ from actors [section 2.2] in many ways. The most significant difference is the principle behind spawning of actor and spawning of isolate. An actor is supposed to be a very lightweight and cheap to spawn but spawning isolates in Dart takes significant amount of time and resource. The implementation of actor found in other languages like Erlang [section 2.3] and Akka toolkit [section 2.5] can be considered much closer to Hewitt's actor model.

The number of actors that can be spawned per GigaByte of heap memory in Akka reaches up to 2.7 millions [Incb] whereas, an isolate in dart takes up around 5 to 7 MegaBytes[6] of memory, the number of isolates per GigaByte of heap can only reaches up to few hundreds. Based on these observations, it would be appropriate to say that the current[7] implementation of an isolate in Dart is — similar to a threads with properties like an actor.

Limitations of Isolates

Even though communication between two isolates takes place exclusively by asynchronous message passing the perfectly suits for distributed systems, there is no implementation of communication with isolate spawned in another Dart virtual machine. A message exchange between two isolates via SendPort/ReceivePort is possible only if the isolates are spawned locally in the same Dart virtual machine. Thus, a hindrance in making them distributed.

2.7 RabbitMQ - A Message Broker System

RabbitMQ is an open-source simple message broker software that implements Advanced Message Queuing Protocol(AMQP). It serves as an intermediary for messaging for applications or components of an application so that they can connect with each other, while keeping them loosely coupled. It give applications a common platform to send and receive messages, and keeps the messages until received [Sofe].

[6] Based on the memory consumption of the two isolates from Listing 2.3

[7] Dart version 1.7.2

Messaging enables software applications to connect and scale. Applications can connect to each other, as components of a larger application, or to user devices and data. Messaging is asynchronous, decoupling applications by separating sending and receiving data. In RabbitMQ, messages are routed through exchanges before arriving at queues. RabbitMQ features several built-in exchange types for typical routing logic [Sofe].

RabbitMQ allows several servers of a local network to form a cluster. The cluster forms a single logical broker and queues are mirrored across several machines, which means the applications that uses it may connect to any of the servers that belong to the cluster [Sofe].

RabbitMQ supports several protocols for enqueuing and dequeuing messages:

- AMQP (Several versions)
  The Advanced Message Queuing Protocol (AMQP) was designed to provide reliability and interoperability. It provides messaging, including reliable queuing, topic-based publish-and-subscribe messaging, flexible routing, transactions, and security. AMQP exchanges route messages based on topic and headers[Pip]. Despite the fact that there are many different language implementations[8] and examples for using AMQP in RabbitMQ, it is still not available for the Dart language.

- STOMP
  STOMP [section 2.8] is a text-based messaging protocol emphasizing simplicity. More about STOMP is discussed in section 2.8.

  RabbitMQ supports STOMP via a plugin – 'rabbitmq_stomp'.

- MQTT
  Message Queue Telemetry Transport was developed by IBM. It provides lightweight publish-and-subscribe messaging, targeted for resource devices with low resources and limited network bandwidth. Hence, the design principles and aims of MQTT are simpler and more focused than those of AMQP. [Pip]

  RabbitMQ supports MQTT 3.1 via a plugin.

- HTTP
  HTTP is not a messaging protocol. Nevertheless, with the help of the listed technologies, that use HTTP as their substructure, RabbitMQ can transmit messages over HTTP [Soff].

[8] Python, Java, Ruby, PHP, C#, Erlang etc. [Sofb]

Management Plugin
: It supports a simple HTTP API to send and receive messages. This is primarily intended for diagnostic purposes but can be used for low volume messaging without reliable delivery.

Web-STOMP Plugin
: The plugin supports STOMP messaging to the browser using WebSockets. For the older browsers that do not have support for WebSockets, fallback mechanisms provided by SockJS[9] is used.

JSON-RPC channel Plugin
: This plugin support AMQP 0-9-1 messaging over JSON-RPC[10]. It is a synchronous protocol, thus the asynchronous delivery property of AMQP is emulated by polling.

2.7.1 Message Queues in RabbitMQ

In RabbitMQ, a queue is a mailbox name, where the messages are stored. It can buffer large quantity of messages bounded only by the available resources of the machine.

RabbitMQ receives messages from a client via one of protocols described in [**??**]. The client specifies the name of the queue, while sending the message, where the message should be enqueued. Meanwhile, the subscribing client of the queue receives a message either as soon as the message is available in the queue, or when the client sends request for dequeue, or only after acknowledging previously dequeued message. This depends upon the parameters used during subscription to the queue.

If inflow of messages in a queue is faster compared to the outflow of the messages to its subscribers, both enqueuing as well as dequeuing gets slower. Since, the messages are buffered into memory, as the quantity of messages increases, the consumption of the memory also increases. Besides, if there is a sudden increase in the inflow of messages, the incoming messages take more CPU time which results in the overall decrease of outflow of messages to consumers [Sofd].

2.7.2 RabbitMQ and prefetch-count

The prefetch-count Quality of Service (QoS) setting in RabbitMQ by default is set to unlimited. Which means, the RabbitMQ empties the queue as fast as possible to the consumer. Setting the prefetch-count to unlimited might result in 'out of memory' or 'stack overflow' errors in the consumers. Again, setting the prefetch-count too low can

[9] SockJS is a JavaScript library that emulates WebSocket in browsers

[10] JSON-RPC is JSON encoded Remote Procedure Call protocol

hamper the performance of the whole application while setting it too high can cause the 'out of memory' exceptions. Hence, based on the requirement and design of the consumer application, appropriate prefetch-count should be determined.

The Figure 2.2 is the result of a benchmark[11] that shows how the throughput of a queue varies when prefetch count is changed for different number of consumers.

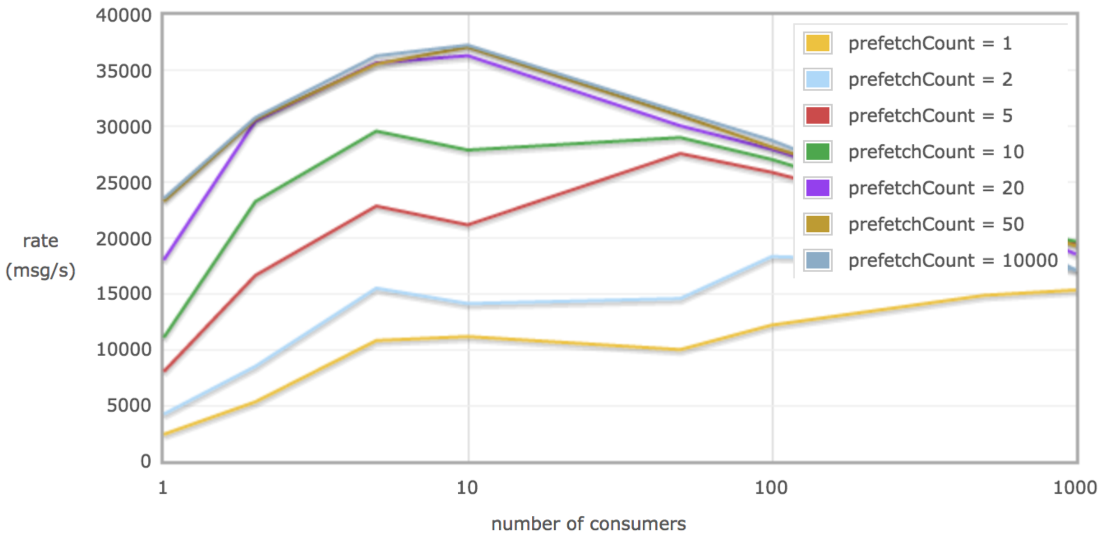

Figure 2.2: Chart[12]showing performance variation when prefetch count is changed in the consumer

2.8 STOMP

STOMP stands for Simple (or Streaming) Text Orientated Messaging Protocol. It is an alternative to other open messaging protocols such as AMQP. It provides an interoperable format for STOMP clients to communicate with a message broker system that supports STOMP. It provides interoperability among different languages, platforms and brokers. [STO] STOMP is designed to be a lightweight protocol that is easy to implement in both client and server.

[11]posted by Simon MacMullen on April 25th, 2012 at 2:47 pm

[12]Source: P. Software. *RabbitMQ Performance Measurements, part 2*. `http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/`. Last Accessed: 2014-11-20

2.8.1 Protocol Overview

STOMP is a frame based protocol. A frame consists of a command, a set of optional headers and an optional body. STOMP is text based but it allows the transmission of binary messages [STO]. A STOMP client can either be producer or consumer. As a producer a message can be sent to a destination to server using SEND frame. Meanwhile, as a consumer, SUBSCRIBE frame should be sent for subscription of a message for a given destination. Then, the messages are received as MESSAGE frames.

2.8.2 STOMP Library for Dart

Given that dart is a fairly new language, there is no AMQP client for RabbitMQ yet. As mentioned above in section 2.7 RabbitMQ also supports STOMP. An open source STOMP client[13] in dart is available created by 'Potix corporation'. It can perform most of the basic operations with message broker system like connecting, creating queue, subscribing, enqueuing and dequeuing. Although it has those basic functionalities, it still has some limitations and incompleteness like lack of support of 'Heartbeat' and it only supports STOMP version 1.2 or above.

2.9 WebSocket

WebSocket protocol enables two-way communication between client and server over a single TCP connection. It uses origin-based security model, which is found in web browsers. It can be used for variety of web applications: games, stock tickers, multiuser applications, user interfaces exposing server-side services in real time, etc. [FM11]

Since, HTTP was not initially designed for bidirectional communication, the WebSocket Protocol is designed to displace other existing bidirectional communication technologies that are based on HTTP [FM11].

WebSocket uses two URI schemes: "ws://" for normal WebSocket connection and "wss://" for secured WebSocket connection [FM11].

2.9.1 Security

The WebSocket Protocol uses the origin model used by web browsers to restrict which web pages can contact a WebSocket server when the WebSocket Protocol is used from a web page [FM11].

[13] https://github.com/rikulo/stomp

A WebSocket server reads the handshake sent by client to establish a connection. Thus, an attempt to connection to WebSocket from other protocols cannot succeed if it is not sent by a WebSocket client. [FM11]

2.9.2 Establishing a Connection

When establishing a WebSocket connection, the HTTP server receives a regular GET request with an offer to upgrade to WebSocket. The server responds to the request to complete the handshake and establish the connection. Then the communication takes place in full-duplex mode [FM11].

3 System Design

3.1 Core Design Decisions

The Framework is designed to be distributed in nature with the concepts of Actors [section 2.2]. It should follow the standard actor programming concept and provide inherently distributed nature to the applications built on top of it. The framework itself should be built using the concept of 'message-passing' [section 2.1] to alleviate any possibility of concurrency issues, thus making the applications thread-safe.

- The framework may not guarantee the delivery of a message.
- All the messages sent by the framework will be based on 'fire and forget' concept.
- A message shall be delivered at most once.
- A message should always be routed through Message Queuing System [subsection 3.2.3], even though the target isolate may belong to same isolate system in the same logical or physical node.
- Feeding of message to an isolate should be based on pull mechanism, not push mechanism.
- Exceptions thrown at child isolates shall be handled by a 'spawner' of that isolate. Hence, implementing the idea of supervision and "let it crash" ideology[subsection 2.3.1].

3.2 The Framework

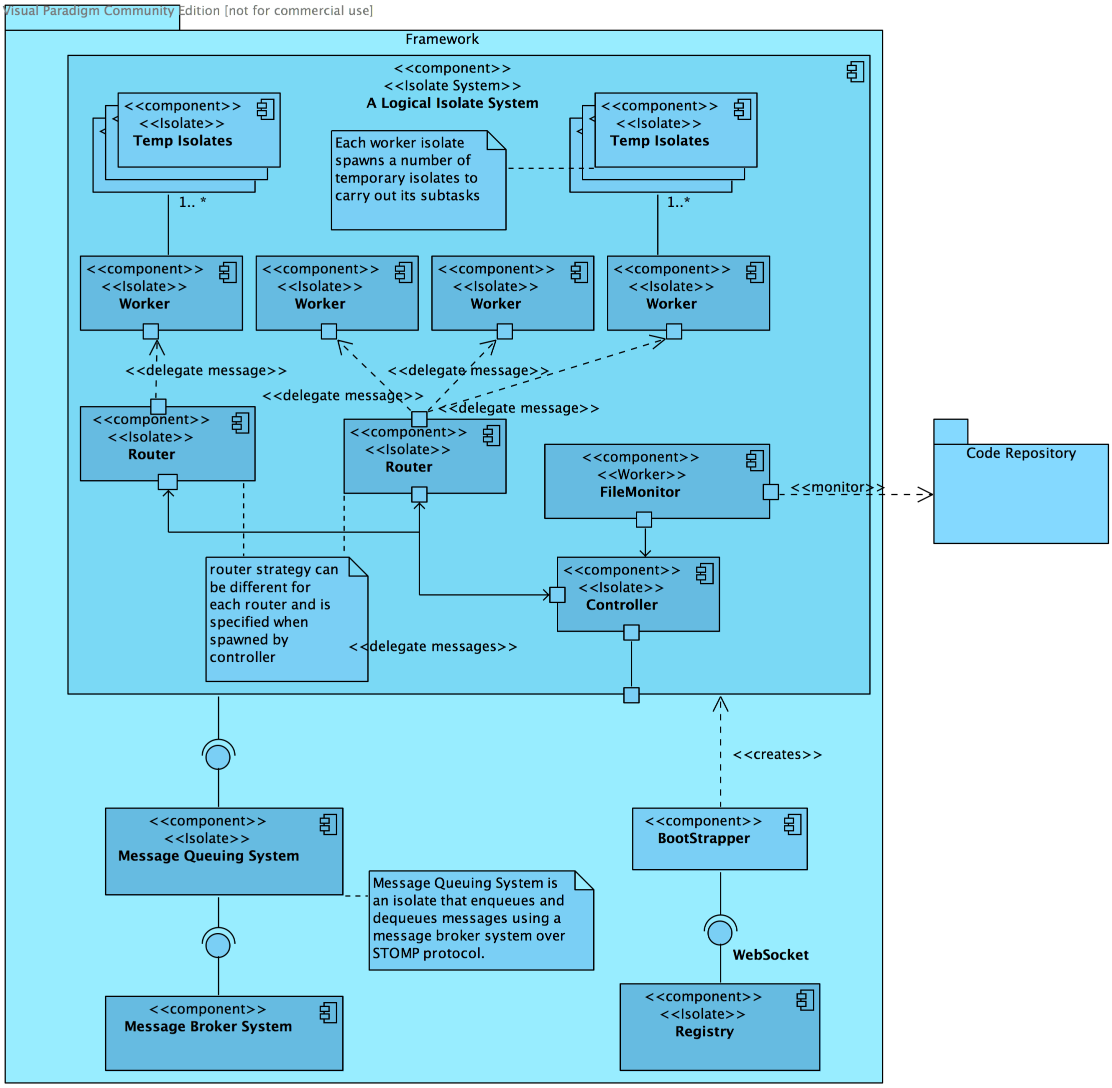

Figure 3.1: Architecture of the framework

The framework comprises of an Isolate System, a Registry, a Message Queuing System, a Message Broker System and an Activator. The Figure 3.1 depicts the general overview of different components and sub-components of the framework.

3.2.1 Isolate System

An Isolate System is analogous to an actor system subsection 2.5.2. Just as an actor system consists of a group of actors working together, an isolate system is composed of a group of 'Worker' isolates working closely. It consists of different hierarchies which forms a logical organizational-like structure. The top level isolate is the Isolate System itself and the bottom most are the 'Worker' isolates.

A 'Bootstrapper' in a physical node can start up several Isolate Systems. Nevertheless, a logical Isolate System is not limited to a single physical node. The 'Worker' isolates spawned by an isolate system can be distributed across several remote systems.

Each isolate system has its unique id, which is a UUID. It is generated when the isolate system is bootstrapped. For bootstrapping, an isolate system needs the WebSocket address of Message Queuing System, and a 'name' for itself. The name is simply an alias, and should not be confused with the unique id as another isolate system with the same name can exist in other nodes but the unique id is exclusive for a particular instance of an isolate system.

The bootstrapping of an isolate system includes: generating a new id which is unique for itself, opening up a 'ReceivePort', and connecting to a 'Message Queuing System'. After opening up a 'ReceivePort', the isolate system starts listening on that port for messages so that it can receive incoming messages from 'Controller'. While connecting to the Message Queuing System, if a connection could not be established, it simply keeps on retrying at certain interval. Furthermore, should the connection be lost at any time after being connected with the MQS, the Isolate System automatically keeps on trying to re-establish the connection at regular intervals. Since, the connection to MQS takes place asynchronously, the isolate system moves forward and spawns a controller, regardless of the establishment of connection with the MQS.

Adding Worker Isolates to The Isolate System

As an isolate system is a top level isolate, it spawns a controller. The controller spawns one or several routers and each router spawns a worker isolates. The Figure 3.2.1 shows the message flow sequence in different components while starting up an isolate system and deploying a worker isolate in it.

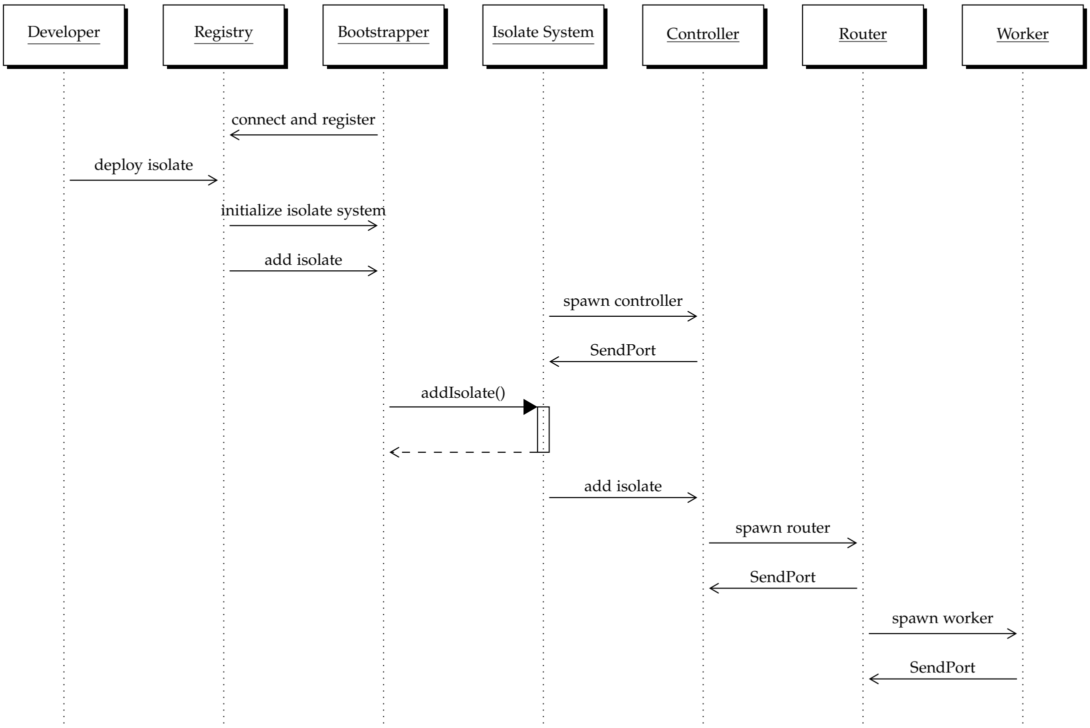

Figure 3.2: Adding a Worker isolate to an isolate system

When an isolate system is first initialized, it is an empty system without any Workers running in it. The Workers can be added with appropriate load balancer once the isolate system has been initialized. The 'addIsolate' method starts up worker isolates into the isolate system. It requires following arguments:

name – A name for pool of isolates. A deployed isolate has its own name but overall the name is concatenated with the name of isolate system to denote the hierarchy. For instance, an isolate with name 'account' becomes 'bank/account' where 'bank' is the name of the isolate system.

sourceUri – The location of the source code from which the isolate shall be spawned. The path can either be absolute path to the local file system or the full http or https URI.

workersPaths – List of destinations where each of the isolates should be spawned. To spawn locally, 'localhost' should be used, whereas to spawn in remote node, WebSocket path like: "ws://192.168.2.9:42042/activator" should be used. The number copy of isolates that should be spawned is determined by the length of this list. If multiple copies of isolates should be spawned in a node, the location can be repeated. For instance ["localhost", "localhost"] results in spawning of two identical isolates in local machine which is load balanced by the type of specified router.

routerType – The type of load balancing technique that one would like to use to effectively distribute incoming messages. By default, the framework, provides three types of routers: Round-Robin, Random and Broadcast. If the developer wants to use his own customized load balancer instead of using the options provided by the framework, the absolute path to the location of source code, which can also be a remote URI, of the custom router implementation can be provided.

hotDeployment – This argument is optional and is set to 'true' by default. Setting it to 'true' enables continuous monitoring of the source code. If any change in source code is identified, the instances of isolates, spawned by this 'addIsolate' function, in current isolate system will be restarted, without the need of redeploying the system.

args – Custom additional arguments to be passed into each instance of spawned isolate. This argument is also optional and can be safely ignored.

Message Handling in The Top Level Isolate System

Typically, a message in an isolate system may arrive from three sources: Message Queuing System via WebSocket, Controller via ReceivePort or Bootstrapper via direct method invocation. As Dart is a single threaded programming language [subsection 2.6.5], only one message is handled at a time.

A Message Queuing System sends the data over WebSocket in JSON string format, which should be deserialized to Map data type before further processing. As the received message contains the queue name from which it is dequeued, the name of the queue is then parsed and transformed to name and address of the corresponding isolate. The message is then forwarded to the Controller that this instance of the isolate system has spawned.

The messages arriving from a controller is either a dequeue requests or a message that should be sent to the Message Queuing System for enqueuing. The dequeue requests are sent from the isolates that have completed certain task and are ready to accept another message. For the dequeue requests, the sender of the message is identified, which is used to figure out the corresponding name of the queue name. Then the pull request to dequeue from that queue is forwarded to MQS via open WebSocket port. For the messages that are supposed to be delivered to another isolate, the name of the target isolate is used to figure out the name of the queue and then sent to the MQS for enqueuing.

The 'Bootstrapper' of a node that creates an isolate system can send messages to isolate system by directly invoking the functions provided by the isolate system. The bootstrapper can request the information about the isolates this instance of isolate system is running. For which, the isolate system delegates the message to its controller and waits asynchronously for the response from the controller. The request, to fetch a list of running worker isolates, is triggered when a user sends the request to view details of an isolate system via a web interface or via RESTful web services provided by the 'Registry' [subsection 3.2.2].

Another type of message is the message to terminate a worker isolate, which is also forwarded to the controller as the isolate system does not directly manage the running worker isolates. Thus, the message is forwarded to the controller which is next in the hierarchy. In contrast, when the shutdown command for the isolate system is triggered via web or REST interface, the isolate system closes all the open ports including WebSocket ports and ReceivePorts, and then wait for the 'Garbage Collector' to and clean up the memory reserved by it.

Controller

Every isolate system has a single controller, which is spawned by the top level isolate of the isolate system. A controller stays idle until it receives a message to create an isolate. Basically, a controller spawns and manages all the routers of an isolate system. Additionally, a controller takes care of the 'hot deployment' feature for which it spawns a 'FileMonitor' for each router if the feature is enabled. When a RESTART message is received from a FileMonitor, the controller sends a RESTART_ALL message to the designated router, which restarts all the Worker isolates the router has spawned.

A controller is also responsible for replying to the query of list of isolates an isolate system is running. It achieves this by keeping a detailed record of each Router and number of Worker isolates each Router is handling, which is updated as soon as an isolate is killed or a new isolate is added.

As a Controller is the 'spawner' of Routers and the 'spawnee' of the top level isolate, it forwards the messages as well as dequeue requests coming from Routers to the top level isolate of an isolate system.

Router

A router is spawned by a controller. The router creates and is responsible for a group of identical Workers isolates. Since an isolate is single threaded, creation of multiple instances of an isolate is desirable for concurrency. When a message arrives in a router from a controller, the router, based on its defined routing policy, delegates the message to one of the worker isolates. The routing policy can be chosen at the time of deployment of a worker isolate.

A router uses a routing policy to distribute message among the group of isolates it is handling. The default routing policies that are available in the framework are listed in the Table 3.1

Table 3.1: List of routing techniques provided by the framework

Router	Description
Round Robin	Messages are passed in round-robin fashion to its Worker isolates.
Random	Randomly picks one of its Worker isolates and sends the message to that Worker isolate.
Broadcast	Replicates and sends message to all of its Worker isolates.

In addition to the available routing policies of the framework, it is also possible to add a new Routing technique by simply extending the 'Router' class which requires 'selectWorker' function to be implemented. The overridden 'selectWorker' function may either return a list of Workers or a single Worker. The ability to implement a custom router opens up possibilities for numerous load balancing techniques. For instance, a simple multicasting router that replicates a message only to the Workers that are spawned locally can be implemented by selecting such Workers using their deployment paths and returning them as a 'List'.

As the router manages the Worker isolates it has spawned, it is responsible for effectively terminating and restarting the Worker isolates. It also buffers the messages that might arrive while the workers are not ready to accept the messages yet; usually, during the creation of Worker isolates and while restarting them.

If a router does not receive any message from a Worker for a certain amount of time, the router sends a PING message to check if the Worker isolate is alive and ready to accept more messages. If the Worker isolate responds with a PONG message, the router sends a request to fetch messages to controller. This mechanism is present in the framework to prevent the 'starvation' for a Worker isolate in case the dequeue message, that might have been sent earlier, could not reach the Message Queuing System because of a network issue or unavailability of MQS.

Worker

The 'Worker' of the framework is an abstract class, which should be inherited by the isolate that the programmer creates. The 'Worker' first unwraps the messages that arrives from the router and retrieves headers from it. 'Sender' and 'replyTo' headers of the message is collected before forwarding message to the child class, that extends this abstract class. By unwrapping the messages that is encapsulated by various headers, the abstract Worker class makes sure that the message is delivered to the target implementation of the Worker isolate immutated and in intended form.

To extend the 'Worker' isolate, one must implement 'onReceive' function which handles incoming messages and carry out the business logic tasks. However, if a task is too complex, the Worker isolate can divide the tasks into subtasks and spawn temporary isolates to carry out those subtasks concurrently. The temporary isolates can be terminated once the subtask has been carried out.

The 'send', 'reply' and 'ask' functions are provided by this abstract Worker class to send a message to another worker isolate. These functions automatically add the information of sender and receiver in the header of the message that is sent out.

Sending a Message To send a message from one Worker isolate to another, the framework provides 'send' function. It takes 'message' and 'address' of the target Worker isolate as its argument. The reply path can also be optionally set, so that the replied message from target isolate is sent to a different worker isolate for further processing. The named parameter[1] 'replyTo' can be used with the address of actor that is supposed to received the replied message; eg:

```
send("A simple text message", "demosystem/printer");
send("Another message", "demosystem/jsonConverter", replyTo: "demosystem/printer");
```

Asking For a Reply Sometimes a Worker isolate might need a reply from another isolate for further processing or before replying to the sender of the message. In such case, the worker isolate can specifically ask the target isolate to reply to this particular instance of worker isolate.

For instance, a sample use case can be, a worker isolate maintaining a connection with a browser via HTTP. In this case, as the port cannot be serialized and passed to other isolates through messages, another instance of similar isolate will not be able to respond to the request made in that connection.

[1]Dart's named paramter feature

Similar to the 'send' function, the 'ask' function takes 'message' and 'address' of the target worker isolate as its argument; eg:

```
ask("current time", "demosystem/timeKeeper");
```

Replying to a Message To reply to a message, the framework provides the 'reply' function. It expects a single argument — 'message', because the respose is sent to the worker isolate specified by the sender. The 'reply' can be used in response to any of 'send' or 'ask' messages; eg:

```
reply("Current time is: $time");
```

Proxy

A 'Proxy' is the special type of a Worker isolate. When a Worker isolate is supposed to be spawned in a remote node, the router instead spawns a Proxy isolate in local node. Once the Proxy isolate is created, it connects to the 'Isolate Deployer' of the remote node where the Worker isolate is intended to be spawned. After establishing connection over a WebSocket with Isolate Deployer of the remote node, the proxy isolate forwards the request to spawn the worker isolate to the Isolate Deployer. After successful spawning of isolate in the remote node, the proxy isolate simply forward the messages that is sent to it by the spawner router. Each proxy worker maintains a separate WebSocket connection with an 'Isolate Deployer'.

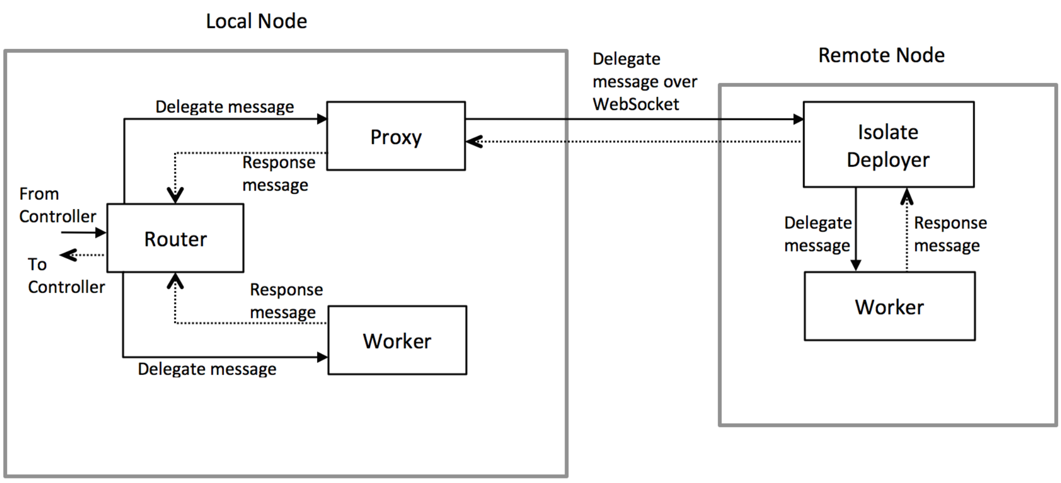

Figure 3.3: A Proxy Worker

FileMonitor

The controller spawns a 'FileMonitor' only if the 'Hot Deployement' flag for an worker isolate is set while deploying. The spawned 'FileMonitor' monitors the md5 checksum of the source code using which the Worker isolate is spawned. If a change in the source file is detected, it simply sends a RESTART command to the controller, which eventually forwards it to the target router. The router then restarts all of its worker isolates.

3.2.2 The Registry

The Isolate Registry is a central node where other nodes, which are running Bootstrapper, connect and register themselves. The registry simply keeps the record of the connected nodes, assigns a unique id to each and queries them about the running isolate systems when required. The registry provides RESTful API and a web interface [2] through which one can have an overview of the full system and manage the deployments of the isolate systems as well as individual isolates.

The basic tasks that a registry carries out are:

- Bootstraps an isolate system, during runtime, in local or remote node
- Provides a way to deploy, update or remove an isolate system
- Returns information about the deployed isolates by querying the individual isolate system of a node.

RESTful API of Registry

The registry provides a REST API to perform the operations on the connected nodes. One can send a 'GET' request to the registry to fetch the list of the nodes that are connected to the registry. Using the 'id' of a node from the replied list, one can deploy an isolate system or add an isolate to already deployed system by sending appropriate 'POST' request.

The REST endpoints exposed by the registry are listed in table Table 3.2.

[2] the web interface can be opted out as it has to be started separately

Table 3.2: List of endpoints exposed by the registry

Method	Endpoint
GET	/registry/system/list
GET	/registry/system/{bootstrapperId}
POST	/registry/deploy
POST	/registry/system/shutdown

A sample GET and POST query to deploy an isolate system

GET request to fetch a list of connected systems

Request: 'GET `http://54.77.239.254:8000/registry/system/list`'

Response Status Code: 200 OK

Response Body:

```
[
  {
    "bootstrapperId": "266393094",
    "ip": "54.77.239.244",
    "port": "50189"
  },
  {
    "bootstrapperId": "12133208",
    "ip": "54.77.239.243",
    "port": "50192"
  }
]
```

POST request to deploy an isolate system

Request: 'POST `http://54.77.239.254:8000/registry/deploy`'

Request Body:

```
{
  "bootstrapperId" :"266393094",
  "action": "action.addIsolate",
  "messageQueuingSystemServer": "ws://54.77.239.200:42043/mqs",
  "isolateSystemName" :"sampleSystem",
  "isolateName" :"consumer",
  "uri" :"http://54.77.239.221/sampleSystem/bin/Consumer.dart",
```

```
  "workersPaths" :["localhost",
      "ws://54.77.239.243:42042/activator"],
  "routerType" :"random",
  "hotDeployment" :true
}
```

Response Status Code: 200 OK

A sample GET query to fetch details of an isolate system

GET request to get details of an isolate system
Request: 'GET `http://54.77.239.254:8000/registry/system/266393094`'
Response Status Code: 200 OK
Response Body:

```
{
  "sampleSystem": [
    {
      "id": "sampleSystem/consumer",
      "workerUri":
          "http://54.77.239.221/sampleSystem/bin/Consumer.dart",
      "workersCount": 2,
      "workersPaths": [
        "localhost",
        ws://54.77.239.243:42042/activator"
      ],
      "routerType": "random",
      "hotDeployment": true
    }
  ]
}
```

A sample POST query to terminate a Worker isolate

POST request to terminate an isolate of an isolate system
Request: 'POST `http://54.77.239.254:8000/registry/system/shutdown`'
Request Body:

```
{
  "bootstrapperId" :"266393094",
```

```
  "isolateSystemName" :"sampleSystem",
  "isolateName" :"consumer"
}
```

Response Status Code: 200 OK

An example of terminating an Isolate System

POST request to terminate an isolate of an isolate system
Request: 'POST `http://54.77.239.254:8000/registry/system/shutdown`'
Request Body:

```
{
  "bootstrapperId" :"266393094",
  "isolateSystemName" :"sampleSystem"
}
```

Response Status Code: 200 OK

The registry generates all the information about isolate and isolate systems "on the fly". Thus, it does not need to persist any data.

The Web Interface for the Registry

The deployment of isolates can also be managed by using a web interface provided by the registry. The Web Interface should be started up separately in a different port. The Web Interface, internally communicates with the registry via the REST API [3.2.2] which is exposed by the Registry.

<TODO: A Screenshot here or in the Appendix>

3.2.3 Message Queuing System (MQS)

Since, the basis of this system is message passing, the Message Queuing System is an important component of this framework. The MQS is a top level isolate that fetches messages from message broker system and dispatches to respective isolate of the isolate system. Whenever a new isolate system starts up, it opens up a new WebSocket connection with the MQS. The messages are exchanged between the isolate system and the MQS through the WebSocket connection. The MQS keeps track of the unique-id of an isolate system so that it can identify the origin of the message.

If a message is supposed to be enqueued, the MQS ignores the unique-id and simply forwards the message to the 'Enqueuer' isolate. Whereas, if the message is a dequeue request, the MQS forwards the the message to a 'Dequeuer' isolate along with the unique-id of the isolate system. The unique-id is used to identify the WebSocket port through which the request arrived. Thus, making sure that the dequeued message is sent to the correct requester. This is especially required if a cluster, of identical isolate systems, is running on different nodes.

The MQS should be started up separately along with few command line arguments to connect to message broker system. The required command line arguments are: ip address, port, username and password to connect to Message Broker System. The 'prefetchCount' is an optional argument which defaults to 1, if not provided explicitly. A 'prefetchCount' is a Quality of Service header for Message Broker System which allows a subscriber of a queue to hold the defined quantity of unacknowledged messages.

Since, in Dart[3] the passing of sockets to isolates is not yet possible, so the main isolate has to pipe all the input/output data. In this case the MQS is the top level isolate which has to handle all incoming and outgoing messages.

Enqueuer

An enqueuer is a separate isolate. A Message Queuing System has only one enqueuer, which basically receives messages from the MQS and sends messages to a message broker system – RabbitMQ [2.7] via STOMP [2.8] protocol.

Dequeuer

As opposed to Enqueuer, a Message Queuing System maintains each dequeuer for each topic. The topic corresponds to each router running in the isolate system. Whenever a message arrives from a new isolate, the MQS spawns a new dequeuer isolate. The dequeuer then subscribes to a new message queue in the message broker system via STOMP [2.8] protocol. If the queue does not exist in the message broker system, the message broker system automatically creates the queue.

If a dequeuer is idle for too long, i.e. if the Dequeuer isolate has not received any dequeue requests for certain interval[4], then the MQS terminates the dequeuer isolate for that particular queue. Nevertheless, as soon as the MQS receives a dequeue request, it spawns a new Dequeuer, if one does not exist yet.

The dequeuer subscribes messages from Message Broker System with such options that the new messages do not arrive to the subscriber unless previously dequeued messages have been acknowledged. This throttles the flow of messages from message

[3] Dart version 1.7.2

[4] by default the timeout is 10 seconds

broker system and keeps itself and the isolates from being overwhelmed by a large number of messages, which might induce 'out of memory' issues.

Messages in dequeuer keeps on arriving as long as there are messages in the queue and the messages are being acknowledged. The messages that are in the buffer of dequeuer stay in unacknowledged state unless they are flushed and sent out to the requesting isolate of an isolate system. As soon as a message is acknowledged the dequeuer receives another message from the message broker.

Multiple Instances of MQS

It is possible for a system to have multiple Message Queuing Systems for scaling up the system. If there are multiple identical isolate systems connected to different instances of MQS, each MQS will have a dequeuer which subscribes to the same queue. Nevertheless, a message is dispatched by the message broker system to only one of the dequeuers, which is distributed in round robin fashion. Thus, messages are fairly distributed among the subscribers.

3.2.4 Activator

An activator simply starts up two isolates: a 'System Bootstrapper' and an 'Isolate Deployer'. Every node that is supposed to be running an Isolate System or become a part of isolate system by running isolates must be running an Activator. The activator requires a WebSocket address of the Registry as a command line argument. Nevertheless, it is also possible to start up System Bootstrapper and Isolate Deployer separately.

System Bootstrapper

The System Bootstrapper registers itself to the 'Registry' via a WebSocket connection as soon as it is started. The activator forwards the path of the WebSocket to the system bootstrapper. But, if a System Bootstrapper is started separately then the path of the Registry should be passed as a command line argument.

Isolate Deployer

An Isolate Deployer starts up a Worker isolate in a node. The isolate is spawned without a local isolate system and as a part of an isolate system running in another node. This functionality expands the isolate system beyond a physical system. An isolate system can deploy number of instances of an isolate in several different nodes.

An isolate deployer running in a remote machine is able to handle requests from multiple 'Proxies' from several isolate systems. Each proxy opens up a separate WebSocket channel with the isolate deployer.

3.3 Some Key Features

3.3.1 Hot Deployment of Isolates and Isolate Systems

It is possible for the source code, of an isolate, to reside in a remote repository and fetched by the controller of a node when required. For instance: isolate source code can reside in a git repository hosted in GitHub. So that as soon as new code is committed in the repository, it gets immediately picked up by the application and the change gets reflected without restarting the application.

After a node is bootstrapped, changes like: addition, update or removal of isolates in an isolate system can take place. In such case, the isolates can be killed and redeployed when it has finished processing tasks and is sitting idle. A dedicated isolate 'FileMonitor' [section 3.2.1] monitors changes in the code repository. When a change is detected, the 'FileMonitor' isolate sends a RESTART message along with the target router to notify the controller. The controller takes care of pushing the message to relevant router, and the router takes care of terminating and re-spawning the worker isolates.

This hot deployment capability improves the availability of an application. Whenever there is any change in a component of an application, the whole application does not need to be re-deployed, instead, only a set of isolates that should be updated is restarted at runtime. This increases overall up-time of the application and keeps other components working even in the time of modifications.

3.3.2 Migration of Isolates and Isolate Systems

Relocation of Worker isolates or an isolate system during runtime i.e. killing a set of Worker isolates or an isolate system at one node and bringing up same set of Worker isolates in another node is the migration of isolates or isolate system. The concept of hot deployment and migration brings enormous possibilities in a distributed system. Some of them are:

- Migration of actors/isolates allows an application to scale in an easy way. With this capability, it is also possible for an application to be brought up the most frequently used isolates near to the server where it is accessed the most.

- Related and dependent isolates can be migrated to the same server, if it is evident that it improves performance of the entire system.
- In case of hardware failure on a system which is running a certain set of isolates, migration of worker isolates during runtime can make the application survive the hardware failure.

3.3.3 Remote Isolates

The current isolate implementation in Dart[5] cannot communicate with other isolates over a network. The Worker isolates in this framework have an ability to communicate with the isolates that may be running in a remote node. So, there can be isolates running in any node. The communication underneath is taken care of by the framework so the implementer of this framework does not have to worry if an isolate is remotely spawned or locally spawned.

Two isolates, although, running in two different virtual machines, can still belong to a same logical isolate system.

3.4 Typical Message Flow in the System

The framework is based on 'fire-and-forget' principle of message sending. The Figure 3.4 shows a simple message flow while enqueuing a message and the Figure 3.5 shows the process of dequeuing a message. A message is serialized to JSON string before sending via a SendPort of an isolate and deserialized after receiving from ReceivePort.

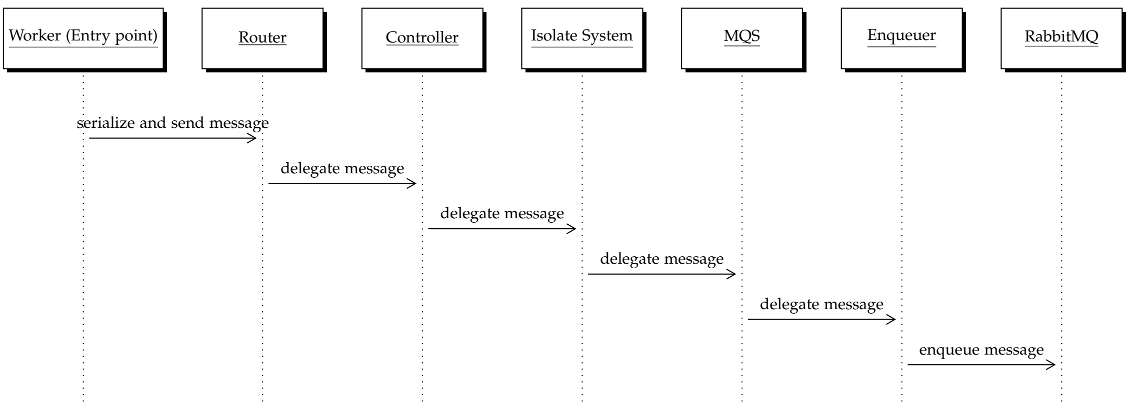

Figure 3.4: Enqueuing a message

[5] Dart version 1.7.2

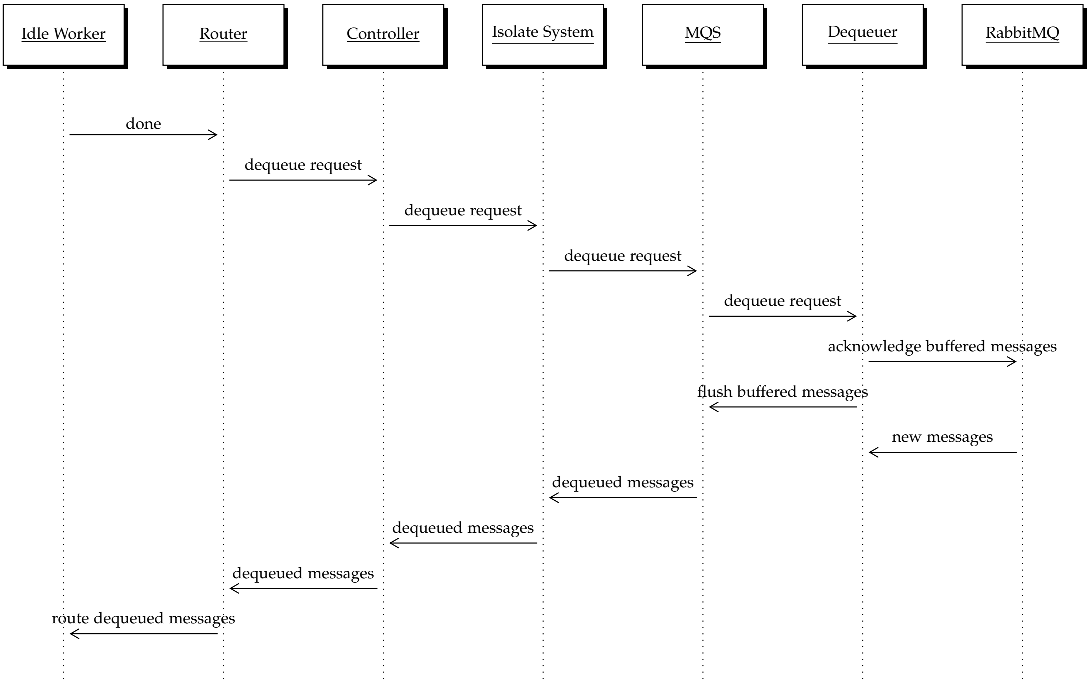

Figure 3.5: Dequeuing a message

3.4.1 Sample message formats

The message sent through different components while enqueuing

Original Message:

```
"Test"
```

Worker:

```
{senderType: senderType.worker, id:
    sampleSystem/producer/88f52440-5060-11e4-f396-97cebb949945,
    action: action.send, payload: {sender: sampleSystem/producer,
    to: sampleSystem/consumer, message: Test, replyTo: null}}
```

Router:

```
{senderType: senderType.router, id: sampleSystem/producer, action:
    action.send, payload: {sender: sampleSystem/producer, to:
    sampleSystem/consumer, message: Test, replyTo: null\}\}
```

Controller:

```
{senderType: senderType.controller, id: sampleSystem/producer,
   action: action.send, payload: {sender: sampleSystem/producer,
   to: sampleSystem/consumer, message: Test, replyTo: null\}\}
```

Top level isolate:

```
{targetQueue: sampleSystem.consumer, action: action.enqueue,
   payload: {sender: sampleSystem/producer, message: Test, replyTo:
   null}}
```

Message Queuing System:

```
{topic: sampleSystem.consumer, action: action.enqueue, payload: {
   sender: sampleSystem/producer, message: Test, replyTo: null}}
```

Enqueuer:

```
{sender: sampleSystem/producer, message: Test, replyTo: null}
```

Sample format of a message sent at different components while dequeuing

Dequeuer:

```
{"sender":"mysystem/producer","message":"Test","replyTo":null}
```

Message Queuing System:

```
{senderType: senderType.dequeuer, topic: mysystem.consumer, payload:
   {sender: mysystem/producer, message: Test, replyTo: null}
```

Top level isolate of isolate system:

```
{senderType: senderType.isolate_system, id: mysystem, action:
   action.none, payload: {to: mysystem/consumer, message: {sender:
   mysystem/producer, message: Test, replyTo: null}}}
```

Controller:

```
{senderType: senderType.controller, id: controller, action:
   action.none, payload: {to: mysystem/consumer, message: {sender:
   mysystem/producer, message: Test, replyTo: null}}}
```

Router:

```
{senderType: senderType.router, id: mysystem/consumer, action:
   action.none, payload: {sender: mysystem/producer, message: Test,
   replyTo: null}}
```

Worker:

```
"Test"
```

3.4.2 Some Implementation Overview

Some insight about the implementation of selected functions of the framework:

How 'send' works?

When a Worker Isolate sends a message using 'send' function of the Worker class, the message is encapsulated with further information about the sender and the receiver are added to the message. The encapsulated message is forwarded to the spawner isolate, which in this case, is the router. The router agains forwards it to the controller which again forwards to the top level isolate. Then the top level isolate adds another level of encapsulation and headers to the message so that the Message Queuing System knows the destination queue.

If the Worker isolate is expecting to consume another message after sending a message, it should send a PULL Request for another message, which can be performed by invoking the 'done' function.

How 'ask' works?

Ask function has certain subtle differences from the 'send' function. The 'ask' function should be used when the sender of the message expects something in reply. The abstract Worker class adds the full path of the isolate along with the unique-id of the isolate when an 'ask' message is constructed. This is to make sure that the response from the target isolate reaches this particular instance of the isolate. The Router, which can also be called a load balancer, when receives a message with full address of a Worker isolate, routes the message to the isolate with the unique-id contained in the message. The message is simply discarded by the Router, if the isolate with the given unique-id is not found in the list of isolates the Router is maintaining. This is possible when the isolates have been restarted or for some reason the isolate was killed.

How 'reply' works?
: The 'reply' function is simply a convenience for the implementer. The 'reply' function simply invokes 'send' message with the sender's address as the target isolate. If the message contains 'replyTo' then the message will be replied to the address contained in 'replyTo' instead of the original sender. The 'reply' function can be used to reply message in both – 'send' and 'ask' cases.

How 'KILL' works?
: This is a special control message sent to the isolates as well as isolate system to shutdown themselves. If a KILL message is sent to a Worker isolate, the message is enqueued to the end of the Worker isolate as any other message. No further messages after KILL a message are sent to that isolate by the router. The router then buffers the messages until the isolate is restarted and resumes sending messages again once the Worker isolate is spawned.

When a Worker isolate finishes processing queued messages and encounters the KILL message, the isolate closes its ReceivePort[6] and stays idle. After sometime it gets 'Garbage Collected' and is cleaned up. But, sometimes the garbage collector cannot clean up the isolate and the memory leak occurs. Thus, as a workaround for this, a custom Exception is thrown deliberately by the isolate to terminate itself. The exception is thrown only after it closes all the ports. This workaround forcefully terminates the isolate and releases the memory occupied by the isolate.

The abstract Worker class provides *beforeKill()* method, which can be overridden to perform custom operations before terminating an isolate.

How RESTART works?
: Restarting an isolate is basically a combined process of killing an isolate and spawning it up again. However, during the restart, after issuing the KILL message, the messages may keep coming from the controller to the router. These messages are buffered in the router itself. The buffered messages are flushed and sent out once the Worker isolates are spawned. For instance, when the 'Hot Deployment'**??** feature is enabled, if the source code of the isolate is modified and saved, the

[6]A Worker Isolate receives message from Router via a ReceivePort

each of the isolates that the router has spawned gets restarted. During which the messages that arrive after RESTART message are buffered in the router.

How shutting down an isolate system works?
An isolate system that is running in a node can be shutdown via the Web Interface or via POST request to the registry. When a request to shutdown an isolate system is sent, the isolate system closes all the ports including the isolate ports as well as the WebSocket connection with Message Queuing System. After that the forceful shutdown is carried out by throwing out a custom Exception. This is a work-around to free up the memory consumed after it is shutdown, because the feature to immediately terminate an isolate is yet to be implemented in dart.

3.4.3 Clustering

Clustering can be achieved in the framework in several levels.

- By deploying worker isolates in several remote nodes. i.e. taking advantage of the concept of 'Remote Isolates' [subsection 3.3.3].
- By deploying replicas of an isolate system in different nodes. An isolate system with same name can exist in another node even though they connect to the same MQS.
- The Message Queuing System itself, can also be replicated where replicas of isolate system may connect to different instance MQS.
- Since, several instances of RabbitMQ [section 2.7] can form a logical group, sharing common configuration, properties, users, queues etc., a cluster of message broker system can be formed. Which allows, Message Queuing Systems to connect to the different member of a cluster.

<TODO: a simple diagram>

3.5 Dart Libraries Used in Construction

Table 3.3: List of libraries directly used by the framework

Library	URL
path	`https://pub.dartlang.org/packages/path`
uuid	`https://pub.dartlang.org/packages/uuid`
crypto	`https://pub.dartlang.org/packages/crypto`
stomp	`https://pub.dartlang.org/packages/stomp`

3.6 A Sample Implementation of Worker Using The Framework

Listing 3.1 is an example of implementation of Worker isolate for the framework. The *main()* is an entry point of the isolate, which is invoked when this isolate is spawned by the router. The class *Consumer* overrides the function *onReceive()*, which is called for each incoming message.

The example shown here prints a message, if the 'action' set in the *message* variable is "print". If the 'action' set in the *message* variable is "send_back" then the Worker isolate sends the message back to the sender of the message.

Listing 3.1: A sample Worker isolate that can be deployed in the framework

```
import 'dart:isolate';
import 'package:isolatesystem/worker/Worker.dart';

main(List<String> args, SendPort sendPort) {
  Consumer printerIsolate = new Consumer(args, sendPort);
}

class Consumer extends Worker {
  Consumer(List<String> args, SendPort sendPort) : super(args, sendPort);

  @override
  onReceive(message) {
    switch(message['action']) {
      case "print":
        print("message['content']");
        break;
      case "send_back":
        reply(message['content']);
        break;
    }
    done();
```

```
    }
}
```

4 Case Study

4.1 A Sample Producer/Consumer Program

4.2 A Sample Requester/Supplier Program

4.3 System Setup for Test

* Used Amazon EC2 instances * A File Server that serves file over HTTP * A RabbitMQ server with one Message Queuing System * 3 other Message Queuing Systems * 32 Nodes

5 Results

5.1 Sample applications made using the framework

5.2 Benchmarks

The benchmark results illustrated in this chapter were performed on Amazon EC2 server[1] with configurations shown in table Table 5.1

Table 5.1: Specification of machines used for testing and benchmarking

Deployed Systems	Name	Specifications
RabbitMQ and a Message Queuing System	c3.2xLarge	8 core CPU, 28 ECU, 15 GiB Memory, SSD disk
Message Queuing System	m3.2xLarge	8 core CPU, 26 ECU, 30 GiB Memory, SSD disk
Registry and File Server	m3.xLarge	2 core CPU, 13 ECU, 15 GiB Memory, SSD disk
Isolate Systems (Nodes)	m3.xLarge	2 core CPU, 13 ECU, 15 GiB Memory, SSD disk

5.2.1 Prefetch Count

Figure 5.1 shows the variation in overall message consumption when the number of consumers (separate isolate systems in separate nodes) were increased. Each line in the figure represents different values of prefetch-count set in the consumer while subscribing to the message broker system. Depending upon prefetch-count, saturation point and decrease rate can also be seen when number of consumers were increased.

The increase in consumers had significant positive result for message throughput, but we can see that after around 8 consumers, adding more consumers had negative effect in the overall throughput.

[1] EC2 servers of Frankfurt data center

Increasing prefetch count had more positive impact in message throughput compared to increasing consumers. For instance, increase in consumers from 1 to 16 resulted in the rise of approximately 2000 message per second, while the increase in prefetch count from 1 to 16 in single consumer resulted in increase of message throughput by approximately 5500 messages per second.

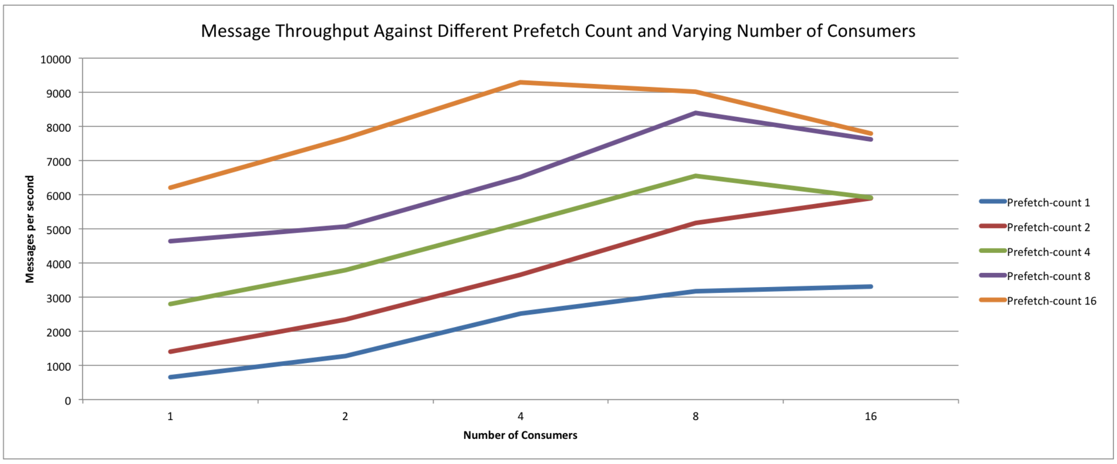

Figure 5.1: Message Throughput vs Number of consumers for varying prefetch-count (Higher is better)

5.2.2 Message Size

The result shown in Figure 5.2 was obtained by allowing eight consumers with the prefetch count of 8 to dequeue existing messages from a queue.

The negative effect, on message consumption throughput, of larger message is evident from the Figure 5.2. The decrease in message throughput was more prominent between message size of 256 bytes and 512 bytes than that of between 64 bytes and 256 bytes.

Similar observation can be made from Figure 5.3 which is a result of allowing a single worker isolate to create messages, except there was a slight rise in the throughput when the message size was increase from 512 bytes to 1024 bytes.

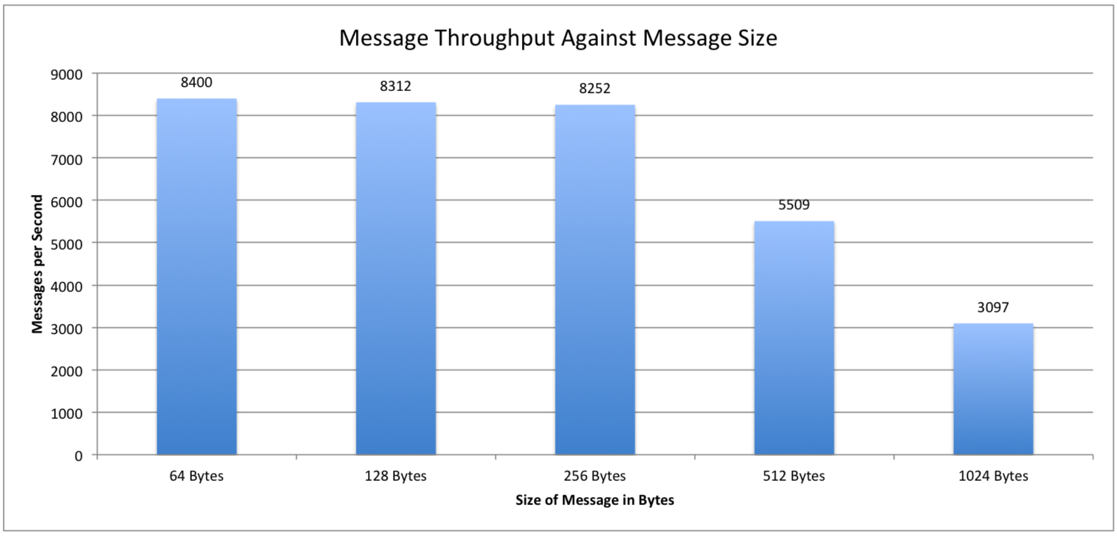

Figure 5.2: Message Consumption Throughput vs Message Size (Higher is better)

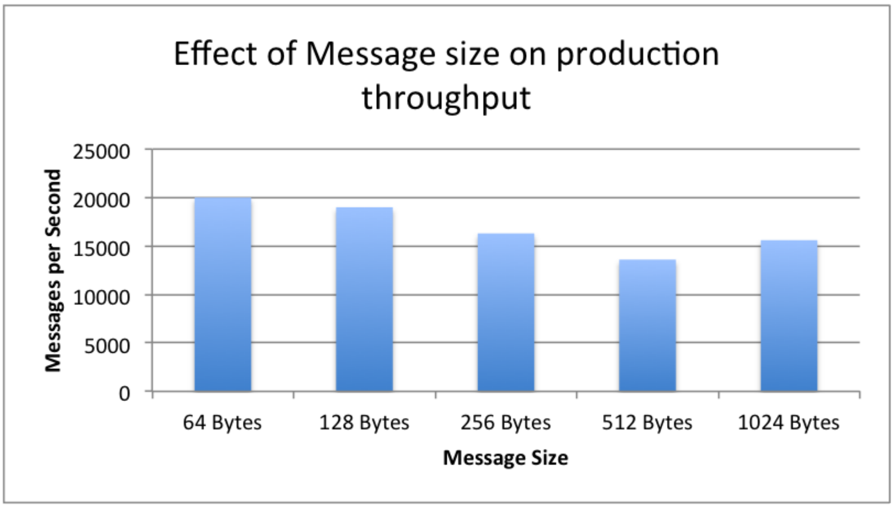

Figure 5.3: Message Production Throughput vs Message Size (Higher is better)

5.2.3 Number of Message Queuing Systems

The Figure 5.4 is the result obtained by testing message consumption by consumers ranging from 1 to 32 distributed to one to four MQS instances connected to same instance of message broker. Multiple parallel instances of MQS clearly had better message throughput than having a single instance of MQS. Nevertheless, adding more than eight consumers per MQS instance had negative effect, as we can see from the fall of throughputs in the line chart of single MQS and that of two MQS. In the tests with one and two MQS the optimum performance was seen when there were 8 consumers in total, but with four MQS, the throughput kept rising and supported upto 32 consumers without fall in the performance. Nevertheless, the rate of performance increase was not as much as the rate seen when scaling up from two MQS to four MQS.

The positive impact of scaling out MQS was seen not only on consumption throughput but also on production throughput. Figure 5.5 shows the result of message production throughput by increasing number of producers from one to eight distributed among multiple instances of instances of MQS. The production rate measured here was the rate at which RabbitMQ enqueued the messages, not the rate at which a worker isolate produced messages.

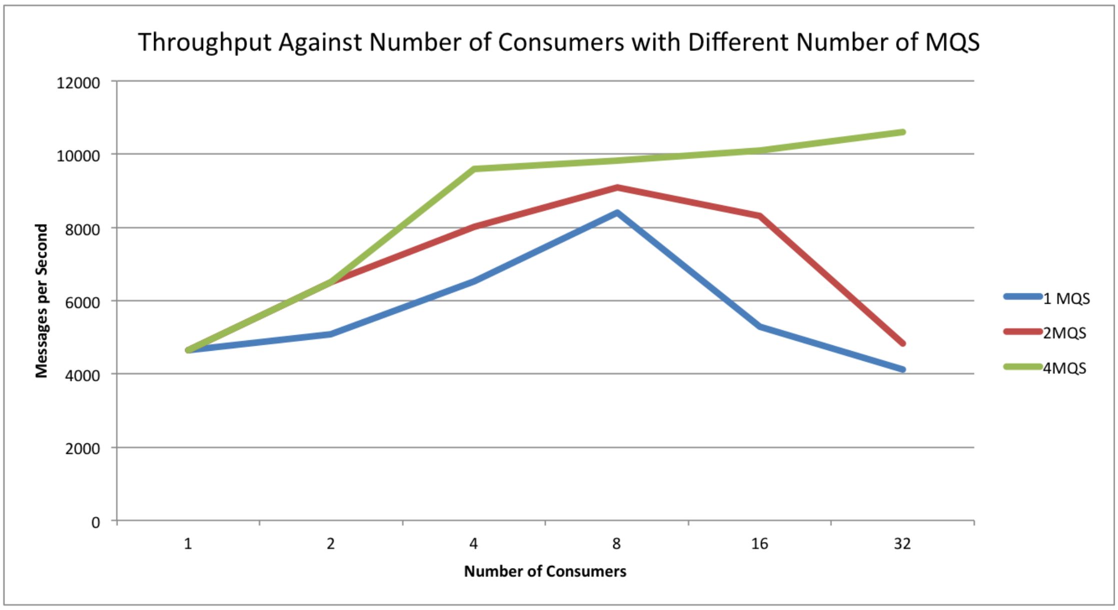

Figure 5.4: Consumption Throughput on Scaled out Message Queuing System

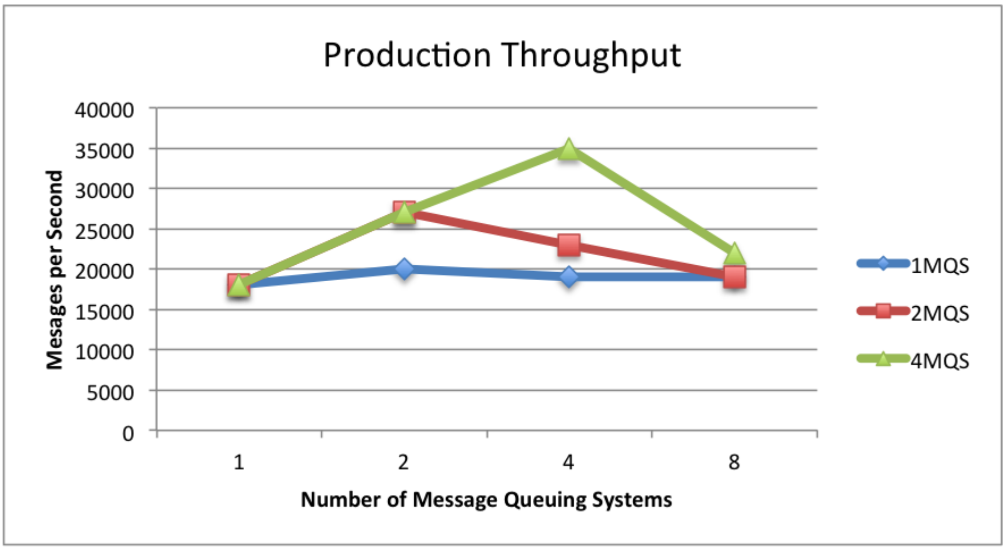

Figure 5.5: Production Throughput on Scaling out Message Queuing System

5.2.4 Production Throughput of Isolates

In contrast to results of production throughput of messages in other measurements as seen on Figure 5.3 and Figure 5.5, the result shown in Figure 5.6 measures the throughput of message production by the isolate system before it is sent to Message Queuing System and to Message Broker System for enqueuing. The production of a message in Worker isolate was throttled, so that there is no immediate 'out of memory' error by overwhelming production of messages. The message production was throttled by delaying the production of messages by random amount of time ranging from 20 micro seconds to 500 micro seconds. The observations made here are the average throughput of each worker isolate as well as average throughput of all the producer nodes connecting to single instance of MQS.

As observed in the Figure 5.6, the total production rate of the messages sharply increased with increase in number of producer nodes. In contrast, the average production of single node was seen lower with higher number of producers.

In Dart version 1.7.2, time required to send a message of 'Map' datatype from one isolate to another consumed around 300-500 microseconds. But, when the same message was serialized to JSON String the time required to send a message dropped to 10 - 40 microseconds. It is an improvement in speed by the factor of 10.

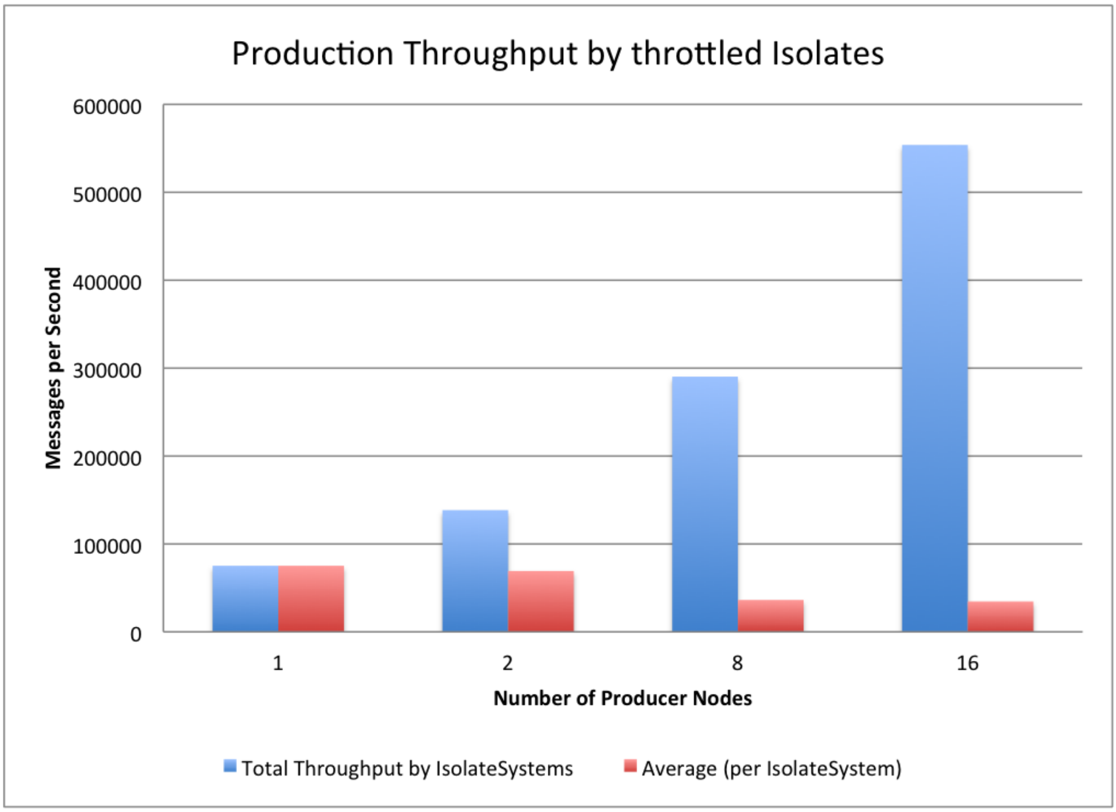

Figure 5.6: Production Throughput of Isolate

5.2.5 Simultaneous Production and Consumption

In contrast to previous benchmarks, which were measured with either only producers or only consumers, the benchmarks Figure 5.7 and Figure 5.8 shows the consumption rate and production rate of messages when they are run simultaneously but in different nodes. Compared to what were seen on Figure 5.1 and Figure 5.4, the consumption rate is lower in this case. Nevertheless, the production rate of messages remained almost equal to that was observed in Figure 5.4 with single MQS.

If we compare the Figure 5.7 and Figure 5.8, the scaling out of MQS with producers in one MQS and consumers in another had significant positive impact in overall message throughput. Especially, in this case the consumption rate increased by almost ten times than that of using single MQS. A slight increase in production throughput was also observed with the additional MQS.

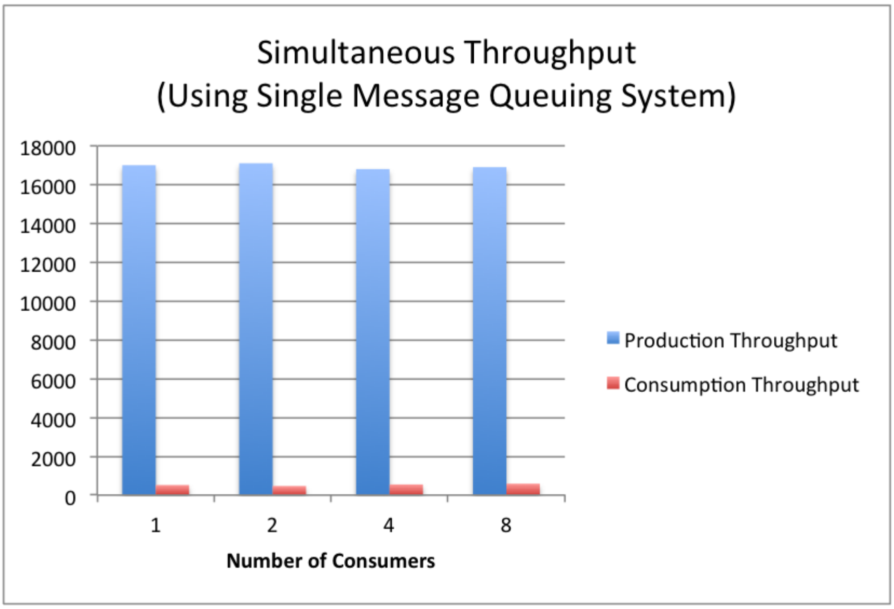

Figure 5.7: Throughput during simultaneous execution in single MQS

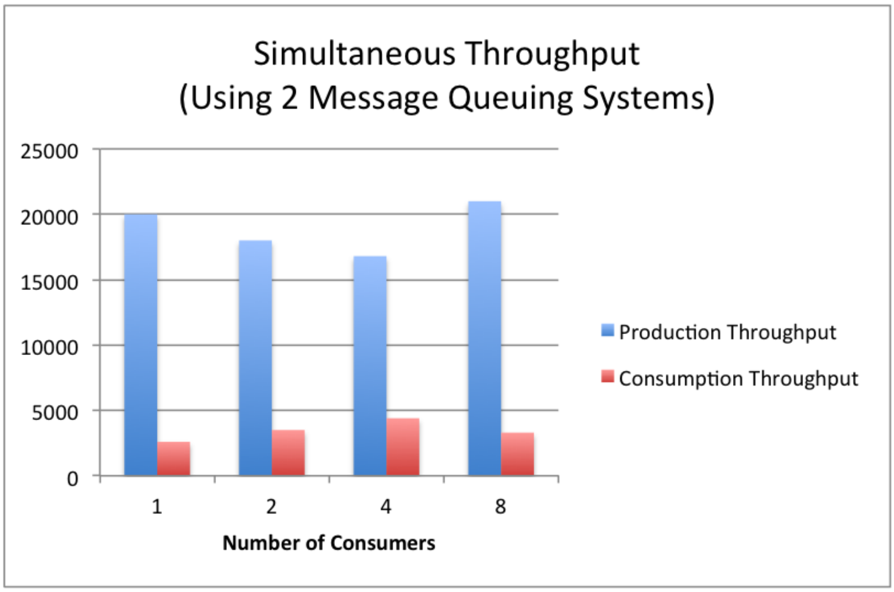

Figure 5.8: Throughput during simultaneous execution in two MQS

5.2.6 Throughput of Request-Reply

The request-reply application used for testing is different from producer-consumer application in terms of how the messages are produced and consumed. In request-reply, a message is produced by the producer only when it receives a request from the requester (and the requester sends request for another message only after it receives the reply) whereas, in producer-consumer application the producer creates messages regardless of existence of consumer. Also, in producer-consumer application any instance of target worker isolate of any node may consume the message as it is designated only by the name of the worker isolate. But, in request-reply the produced message is replied exactly to the same instance of the worker isolate system which sent the request message.

For single execution of a message in producer-consumer, the steps required are enqueuing the message from producer and then dequeuing the message by the consumer. Whereas, in case of request-reply, first a request message has to be enqueued and dequeued, as a reply of which, another message will be enqueued and dequeued. Thus, the steps required in request-reply is twice as much as steps required in the producer-consumer case.

In Figure 5.9, when the number of consumers were increased we can see the linear increment in the production and consumption rate of message. However, adding more suppliers had very little effect in overall throughput. The maximum rise in the throughput by adding suppliers was seen when there were more number of consumers.

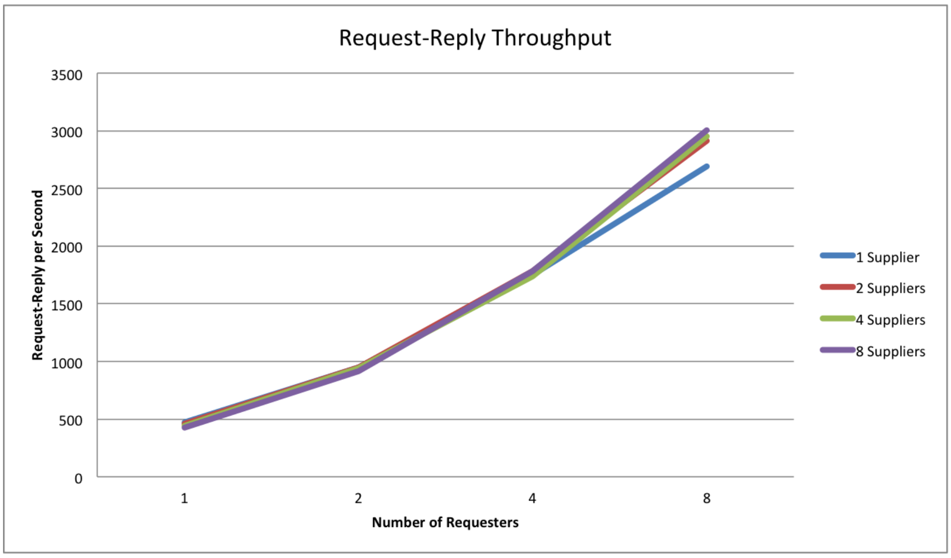

Figure 5.9: Throughput of request-reply

5.2.7 Round Trip Time in Request-Reply

The results seen in Figure 5.10 is the benchmark of average round trip time (RTT) of a message in request-reply case.

The result was obtained by attaching a timestamp in the message that was sent to the supplier; upon receiving the request message at the supplier, the timestamp was copied to the reply message and sent back to the sender(requester). The difference between the timestamp when the message was received at the requester and the timestamp contained in the message is the round trip time.

From the result of RTT in Figure 5.10, we can see that there were inconsistencies in RTT of messages with respect to number of suppliers as well as requesters. When there was only one supplier, the RTT of a message was usually higher compared to cases when there were more number of suppliers. Nevertheless, the least inconsistency was seen in case of 8 suppliers and the least latency was observed in case of single supplier with single consumer.

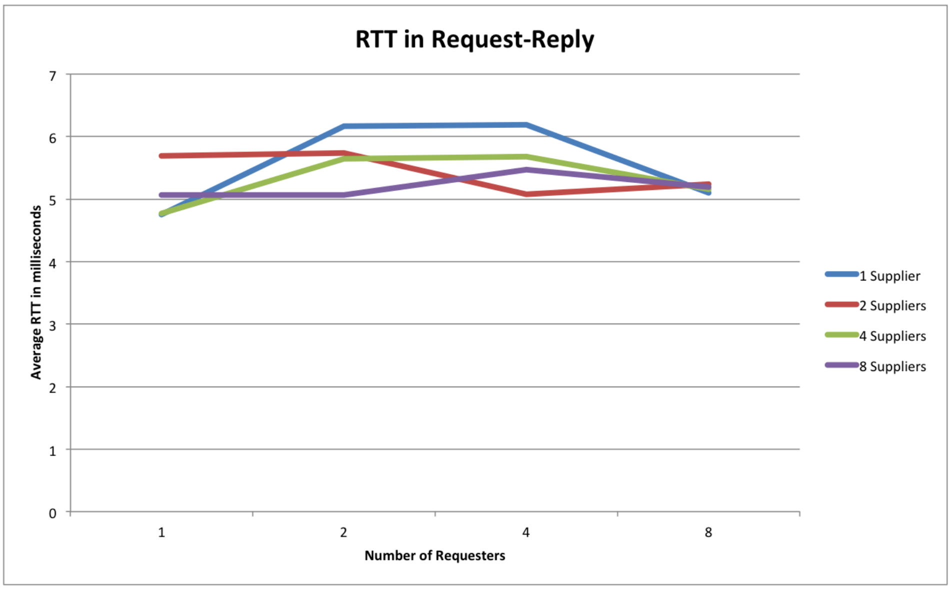

Figure 5.10: Round trip times of messages in request reply (Lower is better)

6 Discussions

6.1 Performance Findings

6.1.1 Effect of Prefetch Count

In the observations made in Figure 5.1, the scaling up of throughput scaled up almost linearly up to 8 consumers for every prefetch-count (except when the prefetch-count was 16). Adding more consumers after 8 resulted in the decrease of throughput, which is quite similar to the result of benchmark [Figure 2.2] of RabbitMQ described in subsection 2.7.2. In the benchmark of RabbitMQ, the decline of throughput were seen when there are around 8 to 10 consumers.

The greater the number of consumers, the more work required by RabbitMQ to keep track of the consumers. Hence, it could be the same reason we saw the decline in performance with more number of consumers in Figure 5.1.

In a distributed system, the cause of a sudden performance change could be because of number of factors like network speed, the system in which the isolate system is running or because of the bottle necking in a Message Queuing System, which is the gateway of all the messages. But each time the performance seems to drop after around 8 consumers. The number of consumers after which the decline in performance occurred in subsection 2.7.2 and Figure 5.1 suggests that the reason could be because of message broker system requiring to handle too many consumers for a queue.

6.1.2 Effect of Message Size

From Figure 5.2 and Figure 5.3 from subsection 5.2.2, we can observe the negative impact of increase in message size. Nevertheless, if we assess the amount by which the performance decreased, we can see that doubling the message size did not cause halving of the performance. Thus, from this finding we can infer that instead of two separate chunks of messages, if we send messages as a big chunk the overall throughput increases. For instance, let's consider the results of 64-bytes messages and 1024-bytes messages. The throughput of 64-bytes messages was 8400 whereas that of 1024 was 3097. If we assume that the 1024-Bytes message was the concatenation of sixteen 64-Bytes messages, then the overall throughput would become 49552 messages per second. The overall increment of throughput by about 6 times! This means that for small messages, time is spent more on processing rather than on Input/Output.

The similar case for production throughput can be seen in Figure 5.3. With similar calculations for 64-bytes messages and 1024-bytes messages, if the messages are chunked together, we can gain approximately 13 times greater throughput.

Nevertheless, such concatenation may not always be feasible and may prove to be against the fair distribution of messages across systems. However, if it is appropriate to concat multiple messages in certain systems, then finding an optimum size (by considering the other requirements of the application) of a message which gives maximizes throughput might prove to be quite beneficial.

6.1.3 Effect of Scaling out Message Queuing System

The scaling out of MQS clearly had positive impact on overall throughput. Higher number of MQS also allowed to have more consumers with good overall performance. The decline of performance seen in tests with 1 MQS and 2 MQS were probably because of the behavior of RabbitMQ when too many consumers are connected as we discussed in subsection 6.1.1. But, the test seen with four MQS suggests otherwise. With four MQS, 32 consumers were supported without drop in performance. From which, we can speculate that MQS might also be causing the limit in the throughput as MQS has to distribute messages to the connected systems. The more the consumers connected to MQS, the more the CPU time and memory it requires.

6.1.4 Comparision of production throughputs

As seen in Figure 5.6 and Figure 5.3, the difference seen between the throughput at a worker isolate and throughput at RabbitMQ is quite significant. This is because the production of message in an isolate is localized to one particular instance before the message is transferred to MQS over WebSocket. Thus, it is natural the production of isolates are that high. Nonetheless, when the overall producers were increased, although overall message production throughput increased, average throughput per isolate declined. One possible explanation to this behavior could be the bottle necking at MQS to which all of these producers send message. The explanation might seem strange because sending messages in isolate are asynchronous. So, it should not have been affected by a outside decoupled component. But, the reason behind this speculation is that since MQS cannot accept messages at the rate which it is produced, the messages get buffered in the internal queues of top level isolates. This takes up more heap memory reserved for the isolate and so it starts to get slower and the similar effect probably occurs in Controller isolate and then in Router isolate and ultimately in

Worker isolate, each of them buffering more messages into internal queue. Hence, as the available heap memory of Worker isolate becomes lower the production throughput decreases.

6.1.5 Effect of Simultaneous Production and Consumption

In Figure 5.7, we can see the negative impact of starting producer along with consumer when producer was producing messages continuously. Again, when the producers and consumers were split to connect to separate instances of MQS, the performance improvement was more prominent in message consumption. By comparing Figure 5.7 and Figure 5.8, the culprit of this behavior seems to be the MQS. The MQS, even though has separate isolates for Enqueuer and Dequeuer, has only one top level isolate subsection 3.2.3. Obviously when there is a large quantity of messages from producer in queue, the dequeue requests that arrive from the consumer isolate as well as the the dequeued messages that arrive from Dequeuer goes further back in the internal isolate queue. The improvement of consumption throughput seen in Figure 5.8 also supports this explanation because in this case, the MQS where consumers are connected does not have to deal with the large surge of production messages. Thus, the dequeue requests and dequeued messages from Dequeuer get processed much quicker compared to previous case.

6.1.6 Throughput of Request-Reply

The benchmark of request-reply subsection 5.2.6 showed almost linear increment with increase in number of requesters. But, increase in number of suppliers from 1 to 8 had little effect in overall performance. The reason behind this could be the design of the program which uses *ask* method to send message. Based on how *ask* works as discussed in subsection 3.4.2, even a single supplier might not have been saturated with the number of requesters it could handle. A slight increase was seen with more supplier, but this is quite minimal and could have been affected by any other factors like RabbitMQ performance, change in network latency, etc. There is not enough data and evidence to support the idea of big improvement in throughput with more suppliers. Testing with more number of consumers till the supplier saturate would yield better result and better idea of what affects throughput of request-reply.

6.1.7 Round Trip Time of Request-Reply

The observation seen in Figure 5.10 was quite inconsistent. It is difficult to make any strong conclusions from the result. Nevertheless, it seems that the increase in number of suppliers decreases latency, as seen in case of eight suppliers against the single supplier.

There could be many factors affecting the result. Even the slight change in network latency can induce a significant change in time required to transfer a message.

6.2 The Framework

6.2.1 Scalability

Scalable at different component levels. Isolates, Isolate System, Message Queuing System and clustering of Message Broker (RabbitMQ). Better scalability is obtained when the Message Queuing System is scaled out.

6.2.2 Availability

Can be designed to make highly available. Apparently, no downtime to deploy new code to a running system.

6.2.3 Reliability

6.3 Problems/Issues

6.3.1 Dart Induced Issues

- The Unimplemented functionalities of Isolates - KILL, PAUSE, PING, Supervision of Isolates
- Isolate are not lightweight.
- Running Isolates in web (Dartium)

7 Conclusion

7.1 TODO

8 Future Directions

Some notes for future directions: * If a code upgrade fails, the isolate should be reverted to its previous state. * Lightweight isolates -> future dart impl? * Detect all sort of exceptions in child isolate -> which would make it possible to implement supervisors and let it crash philosophy. -> future dart impl?, should be available in near future in dart. *

8.1 TODO

9 Appendices

9.1 TODO

Acronyms

URI Uniform Resource Identifier.

UUID Universally Unique Identifier.

List of Figures

List of Tables

Bibliography

[Agh85] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. English. 1985, pp. 34–35.

[AH85] G. Agha and C. Hewitt. “Concurrent programming using actors: Exploiting large-scale parallelism.” English. In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by S. Maheshwari. Vol. 206. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1985, pp. 19–41. ISBN: 978-3-540-16042-7. DOI: `10.1007/3-540-16042-6_2`.

[Arm07] J. Armstrong. “A History of Erlang.” In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, 2007, pages. ISBN: 978-1-59593-766-7. DOI: `10.1145/1238844.1238850`.

[Arm10] J. Armstrong. “Erlang.” In: *Commun. ACM* 53.9 (Sept. 2010), pp. 68–75. ISSN: 0001-0782. DOI: `10.1145/1810891.1810910`.

[Cel11] J. Celko. *Joe Celko’s SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 2011. ISBN: 978-0-12-382022-8.

[ECM14] ECMA. *ECMA-408: Dart Programming Language Specification*. 2014. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), June 2014.

[Erb12] B. Erb. “Concurrent Programming for Scalable Web Architectures.” Diploma Thesis. Institute of Distributed Systems, Ulm University, Apr. 2012.

[FM11] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455 (Proposed Standard). Internet Engineering Task Force, Dec. 2011.

[Goo] Google. *Dart API Reference*. `https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:isolate.Isolate`. Last Accessed: 2014-11-19.

[Hal] P. Haller. *Scala Actors*. `http://www.scala-lang.org/old/node/242`. Last Accessed: 2014-11-19.

[HO09] P. Haller and M. Odersky. “Scala Actors: Unifying Thread-based and Event-based Programming.” In: *Theor. Comput. Sci.* 410.2-3 (Feb. 2009), pp. 202–220. ISSN: 0304-3975. DOI: `10.1016/j.tcs.2008.09.019`.

[HT] P. Haller and S. Tu. *Scala Actors API*. `http://docs.scala-lang.org/overviews/core/actors.html`. Last Accessed: 2014-11-30.

[Inca] T. Inc. *Actor Systems*. `http://doc.akka.io/docs/akka/snapshot/general/actor-systems.html`. Last Accessed: 2014-12-01.

[Incb] T. Inc. *Akka Toolkit*. `http://akka.io`. Last Accessed: 2014-11-19.

[Incc] T. Inc. *Akka Toolkit*. `http://doc.akka.io/docs/akka/2.3.7/AkkaJava.pdf`. Last Accessed: 2014-11-19.

[Kat12] S. L. Kathy Walrath. *Dart: Up and Running*. Last Accessed: 2014-11-19. O'Reilly Media, Oct. 2012. ISBN: 978-1-4493-3084-2.

[Lad] S. Ladd. *Dart Is Not the Language You Think It Is*. `http://radar.oreilly.com/2013/05/dart-is-not-the-language-you-think-it-is.html`. Last Accessed: 2014-12-01.

[Nin] C. Ninners. *The Actor Model (everything you wanted to know, but were afraid to ask)*.

[Nor] P. Nordwall. *Running a 2400 Akka Nodes Cluster on Google Compute Engine*. `http://typesafe.com/blog/running-a-2400-akka-nodes-cluster-on-google-compute-engine`. Last Accessed: 2014-12-01.

[OR14] M. Odersky and T. Rompf. “Unifying Functional and Object-oriented Programming with Scala.” In: *Commun. ACM* 57.4 (Apr. 2014), pp. 76–86. ISSN: 0001-0782. DOI: `10.1145/2591013`.

[Pip] A. Piper. *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP*. `http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html`. Last Accessed: 2014-11-22.

[Sofa] U. o. B. Software Languages Lab. *Concurrent Programming with Actors*. `http://soft.vub.ac.be/amop/at/tutorial/actors`. Last Accessed: 2014-11-30.

[Sofb] P. Software. *RabbitMQ Performance Measurements, part 2*. `http://www.rabbitmq.com/getstarted.html`. Last Accessed: 2014-12-02.

[Sofc] P. Software. *RabbitMQ Performance Measurements, part 2*. `http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/`. Last Accessed: 2014-11-20.

[Sofd] P. Software. *Sizing your Rabbits*. `https://www.rabbitmq.com/blog/2011/09/24/sizing-your-rabbits/`. Last Accessed: 2014-11-22.

[Sofe] P. Software. *What can RabbitMQ do for you?* `http://www.rabbitmq.com/features.html`. Last Accessed: 2014-11-20.

[Soff] P. Software. *Which protocols does RabbitMQ support?* `http://www.rabbitmq.com/protocols.html`. Last Accessed: 2014-11-20.

[STO] STOMP. *STOMP - Simple Text Oriented Messaging Protocol*. `http://stomp.github.io/stomp-specification-1.2.html`. Last Accessed: 2014-11-20.

[Vin07] S. Vinoski. "Concurrency with Erlang." In: *Internet Computing, IEEE* 11.5 (Sept. 2007), pp. 90–93. ISSN: 1089-7801. DOI: `10.1109/MIC.2007.104`.