FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

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 in Informatics

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. Dr. rer. pol. Hans Arno Jacobsen
Advisor:	Richard Billeci
Submission Date:	December 15, 2014

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, December 15, 2014   Sushil Man Shilpakar

Acknowledgments

Abstract

A framework that allows developers to create concurrent, scalable and fault-tolerant applications with high availability in the Dart language is developed in this thesis.

The *isolate* of the Dart language is an interesting entity inspired by the actor model. Their nature of having no shared access to memory and relying on messages for communication makes them asynchronous and decoupled in nature. Nevertheless, their limitation of being able to be spawned only in a local Dart virtual machine restricts them from being distributed. The DDE (Dart Dart Everywhere) framework built as a result of this thesis uses the advantages of the Dart language enhancing it with libraries for remote management, hot deployment of code and distributed execution. The framework provides its own implementation of actors and allows developers to build applications that are inherently distributed in nature.

Contents

1 Introduction

Performance expectations from applications are higher than ever with demands for data and transfer speed increasing rapidly. Performance and scalability concerns are met by using high end multi-core servers and concurrent processing via multi-threading. To scale out and handle a high number of requests, applications are deployed in distributed environments.

Enterprises want applications to be available 100% of the time, because a few seconds of downtime may cause a huge loss in revenue and customer satisfaction.

Ideally an application should be able to scale up or down without compromising its availability. The possibility to dynamically scale allows an application to grow when demand increases and shrink to minimal resources when demand is low.

Deploying applications to distributed environments is a good solution for scaling out, but most applications fail to utilize all available physical resources because of their underlying design.

The actor programming model [section 2.2] allows developers to build highly concurrent, distributed and fault tolerant event-driven applications. These characteristics are provided by actors which use asynchronous message passing [section 2.1]. Actors were first created in the 1970s, but only recently have become mainstream.

The work presented in this thesis focuses on creating a framework for the Dart language [section 2.6] based on the actor-like nature of Dart isolates [subsection 2.6.4], which communicate solely by message passing. The framework takes advantage of isolates and extends their functionalities so that they can be deployed in distributed systems.

The framework is intended to take advantage of the fact that the Dart virtual machine can be run in browser and server. This advantage opens up the possibility of creating a fully distributed application in which isolates may run everywhere: in servers, desktop browsers and even in mobile browsers.

Erlang [section 2.3] and Scala [section 2.4] are popular programming languages which have their own implementations for actor programming. The Akka toolkit [section 2.5] provides a foundation for actor programming in both Java and Scala. The DDE framework presented in this thesis draws on ideas from the actor system [subsection 2.5.2] and router [subsection 2.5.6] implementations of Akka to provide a comparable isolate system [subsection 3.2.1] and routers [section 3.2.1] for Dart.

As DDE applications communicate internally by message passing, the DDE framework takes advantages of features offered by the message broker system – RabbitMQ [section 2.7]. RabbitMQ provides message persistence and a decoupled nature to applications built on the DDE framework.

To provide a fully distributed nature to applications the DDE framework allows developers to deploy components of their application (Worker isolate [section 3.2.1]) to remote nodes. The DDE framework internally uses asynchronous message passing. It has several decoupled components which communicate with each other via WebSockets [section 2.9].

The result of this thesis is a framework based on the Dart language that enforces actor based programming in Dart and makes applications scalable, highly available, and supports deploying and terminating isolates at runtime without interrupting the whole system.

Chapter 2 consists of the introduction and references of the concepts behind the DDE framework. It also discusses the technologies that were used in the framework. Chapter 3 presents the implementation ideas and proof of concept of Dart Dart Everywhere. Chapter 4 shows sample applications built using this framework and presents performance benchmarks and evaluations of applications ran in a distributed environment in Amazon EC2. The deductions made from the benchmarks and results are discussed in chapter 5.

2 Literature Review

2.1 Message Passing Paradigm

Message passing is sending a message from one process, component or actor to another. Message passing is the loosest form 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 the message passing paradigm are easily scalable and efficient. Such systems are easy to replicate and make fault-tolerant. [Arm10].

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 easy to understand and simplifies isolation of problems within the system [Arm10].

Message passing can be synchronous or asynchronous. Synchronous message passing is a communication between two components where the sender blocks until it gets a response from the receiver. 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, asynchronous message passing is non-blocking in nature.

When a message is received by a recipient, the recipient chooses further processing based on the pattern or content of the message.

2.2 Actor Programming Model

The actor 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 [Nin]. 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 for explicit locks. An actor can send messages to actors in the same actor system or another actor system. An actor can also send messages to itself, which is how recursion is achieved. An actor can send a message to a target actor only if it has its address. 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 before it accepted the message

- the target became known when the message was accepted because its address was contained in the message

- the target is a 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 the mailbox is also an actor. According to Hewitt, the order in which messages are delivered is non-deterministic [Nin].

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

- Create other actors

- Send messages to itself, other known actors or reply to the actor that sent the message

- Designate how it is going to handle the 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, where messages are 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 group of actors working together in a certain hierarchy called an *actor system*.

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

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

2.2.1 Error Handling in the Actor Model

Error handling in the actor model is based on the idea of embracing failure. The idea of embracing failure is also known as the "let it crash" philosophy [subsection 2.3.1]. As actors do not have shared state, this allows individual actors to fail without causing disruption in the system. Since an actor in an actor system is typically organized in a hierarchy, 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" philosophy is a non-defensive programming style, which is implemented in Erlang [section 2.3] and Akka.

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. When two or more threads have shared memory, concurrent modification and access of data may result in undesired behavior of the system, known as a 'race condition'. To prevent this type of situation, thread-based programming languages use locks. Locks restrict more than one thread to execute a section of program code [Sofa] concurrently.

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

Actors Based Concurrency

Concurrency in actor based programming languages is inherent because of asynchronous message-passing, pipelining, and dynamic creation of actors. Concurrency in

actors through pipelining is only constrained by logical limits and available hardware resources [Agh85]. Actors may carry out their activities in parallel, as each actor resides in a completely separate space and interacts only via messages.

Actor based programming liberates programmers from delving into coding details about parallelism and threads [Agh85].

'Race conditions', as discussed in section 2.2.2, do not arise in actor based programming, as actors do not have shared state and do not need locks.

2.2.3 Scaling Actor Based Systems

As actor based systems are highly concurrent [section 2.2.2], it is easy to scale them from a single core system to a cluster of multi-core systems, distributed across the globe. An ideal case for scaling would be to let a new node join a cluster and dynamically expand an actor system with zero downtime. The actors themselves do not need to know the physical location of other actors as they simply exchange messages based on logical addresses. This makes them easy to scale up by adding more actors to the same machine and scale out by adding more nodes to a cluster.

2.3 Erlang

Erlang is the first popular programming language 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 and 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. [Arm07]

Erlang was designed for 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 actor model proposed by Agha and Hewitt [section 2.2]. Thus, programs written in Erlang are concurrent, distributed, fault tolerant and thread safe. Concurrency is built into the language [Arm10].

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

Error handling in Erlang is different from most other programming languages. 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 of the "Let it Crash" philosophy is to let failing processes crash, having supervising processes detect and fix the crashes [Arm10]. Error handling in Erlang and the supervision strategy in Akka section 2.5 are based on this principle.

This idea is in sharp contrast to other programming languages, where programmers implement exception handlers and prevent a process from being terminated.

Proponents of the "let it crash" philosophy argue that it leads to more clear and compact code [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 provides a concurrent programming model based on actors [section 2.2]. Actors in scala are fully interoperable with ordinary virtual machine threads [HO09]. Scala actors are lightweight and support approximately 240 times more actors to run simultaneously compared to virtual machine threads [HO09].

Both synchronous and asynchronous message passing may be used in Scala actors. Synchronous message passing is implemented by exchanging several asynchronous messages. Actors in Scala can also communicate using 'futures'. When a future is used requests are handled asynchronously and the sender immediately gets a representation of the future which allows the 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-driven applications on the JVM (Java Virtual Machine) [Incc]. Akka uses lightweight actors for concurrency. Actors in Akka are based on Hewitt and Agha's model [Agh85; AH85]. 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 easily scaling out. Akka is available for Scala as well as Java.

2.5.1 Actor Model in Akka

Akka actors are objects which encapsulate state and behavior, communicating only by message passing, similar to Hewitt and Agha's model [section 2.2]. As Akka actors are objects, they enforce a stringent form of object-oriented programming [Inca].

The 'ActorRef' in Akka represents an actor and holds a reference to an actor. Its purpose is to facilitate message sending to the actor it references. An actor can also refer to itself using *self* [Incb].

Listing 2.1 [2] is an 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. 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 [Incc]

```java
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 arranges actors in a hierarchy. The hierarchy of actors are formed by an actor creating child actors to handle certain tasks. An actor not only creates and assigns tasks to child actors but also supervises them [subsection 2.5.5]. Thus it is implicit that there can be only one supervisor of an actor.

By splitting up tasks, they become clear and structured. Furthermore, the resulting actors become simplified and specialized in terms of which messages they should process and how they should react. It also becomes easier to handle failures. If an actor cannot cope with a failure, it is escalated up the hierarchy to its supervisor. The escalation is repeated by each parent actor in the hierarchy until it reaches an actor that can handle the failure [Incb].

An actor instance in Akka takes up roughly 300 bytes of memory, allowing the spawning of millions of them in one actor system. Akka is able to manage the order of message processing even in large systems with millions of actors [Incb].

An example of the arrangement of actors in a hierarchy is shown in Figure 2.1. The '\' is known as a "root guardian" and \user is known as a "user guradian". Any actor created by a user falls under a user guardian. For instance, in the Figure 2.1, *actorA* and *actorB* are actors created by a user, hence their supervisor is \user. Again, the actors *actor1* and *actor2* are child actors of *actorB*. The address of an actor in Akka is arranged like a filesystem hierarchy. Hence, supervision hierarchies and actor paths are implicit from addresses.
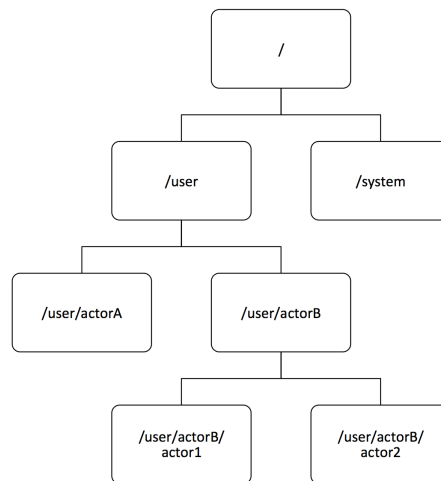
Figure 2.1: Hierarchy in actor system [Incb]

2.5.3 Message Passing in Akka

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

Akka guarantees the order of direct message delivery between two actors. It is only applicable when the mailbox implementation in a receiver is a 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. 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 are delivered before the messages from A3, even if all of the messages from A1 were sent before A3 started sending messages [Incb].

The Akka Java documentation mentions rules for message sending: [Incb]

- at-most-once delivery, i.e. no guaranteed delivery

- message ordering per sender–receiver pair

'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 JVM, there are pitfalls a programmer should know to avoid. Messages used to communicate between actors should be immutable. If messages are mutable and are mutated by an actor in the same JVM, bugs such as race conditions and even unpredictable behaviors might appear. Also, the sender method must not be closed over if the block of code could be run in another thread; for instance, replying 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 the sender may change over time [Incb].

2.5.5 Actor Supervision and Monitoring

A supervising actor in Akka delegates tasks to its subordinates and monitors them for failure. When a child actor detects a failure, it suspends itself and its child actors

and sends a message to its supervisor about the failure. According to Akka Java documentation[Incb], a supervisor may opt to perform one of four choices upon receiving a failure:

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 its 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 remain safe and are processed after the restart [Incb].

2.5.6 Routers

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

- 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 a custom implementation of a router by extending the *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. Location transparency allows Akka applications to be developed locally but deployed into distributed systems simply by changing configurations [Incb].

Peer-to-peer communication between actor systems is the basis for Akka Clustering. Akka's Documentation for Java [Incb] lists 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.

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

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

```
1  // ------------------------------
2  // config on all machines
3  akka {
4    actor {
5      provider = akka.remote.RemoteActorRefProvider
6      deployment {
7        /greeter {
8          remote = akka.tcp://MySystem@machine1:2552
9        }
10     }
11   }
12 }
13
14 // ------------------------------
15 // define the greeting actor and the greeting message
16 public class Greeting implements Serializable {
17   public final String who;
18   public Greeting(String who) { this.who = who; }
19 }
20
21 public class GreetingActor extends UntypedActor {
22   LoggingAdapter log = Logging.getLogger(getContext().system(), this);
23
24   public void onReceive(Object message) throws Exception {
25     if (message instanceof Greeting)
26       log.info("Hello " + ((Greeting) message).who);
27   }
```

---

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

```
28  }
29
30  // -----------------------------
31  // on machine 1: empty system, target for deployment from machine 2
32  ActorSystem system = ActorSystem.create("MySystem");
33
34  // -----------------------------
35  // on machine 2: Remote Deployment - deploying on machine1
36  ActorSystem system = ActorSystem.create("MySystem");
37  ActorRef greeter = system.actorOf(Props.create(GreetingActor.class),
        "greeter");
38
39  // -----------------------------
40  // on machine 3: Remote Lookup (logical home of "greeter" is machine2,
        remote deployment is transparent)
41  ActorSystem system = ActorSystem.create("MySystem");
42  ActorSelection greeter =
        system.actorSelection("akka.tcp://MySystem@machine2:2552/user/greeter");
43  greeter.tell(new Greeting("Sonny Rollins"), ActorRef.noSender());
```

2.5.8 Clustering in Akka

Akka does not have a central server. Instead, it uses a 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. 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 a node to a cluster of at least 1500 nodes took around 15 to 20 seconds on 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]. The Dart language is inspired by Smalltalk, Strongtalk, Erlang, C# and JavaScript [Lad].

---

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

Dart code is not compiled before running; it is read and executed directly from source code by the Dart Virtual Machine (VM). 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 a 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 the Dart Virtual Machine (VM) can be embedded in browsers.

Dart has an automatic garbage collection system, which means the memory occupied by objects which are not in use and which have no references are reclaimed periodically.

Some of its important features and advantages are:

- Easy to learn syntax

- Dart supports types, but they are optional

- Scales from small scripts to large and complex applications

- Supports safe concurrency with isolates

- Supports code sharing

- Translates to JavaScript so that code can be run in web-browsers that do not have the Dart VM yet

- Unified development for client and server

- Runs in client as well as on server

- Has been gaining popularity and adoption in recent years

2.6.2 Dart and JavaScript

Code written in Dart can be translated to JavaScript using a tool – 'dart2js'. 'Dart2js' is bundled with the Dart SDK (Software Development Kit). As popular web browsers like Mozilla Firefox, Google Chrome and Safari do not have a Dart Virtual Machine embedded, the ability to translate source code from Dart to JavaScript lets Dart programs run in any modern browser without needing to manually port the source code to JavaScript.

2.6.3 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 or data from ports.

2.6.4 Isolates

Although Dart programs are single threaded, concurrency is supported using 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 [ECM14].

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

In Dart, when an isolate is spawned, usually a 'sending port' is sent as the initial message (by the spawner), so that the spawner and "spawnee" can communicate with each other. The "spawnee" uses the sending port sent by the spawner to reply.

An isolate can spawn another isolate which can further spawn 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 run on multi-core CPUs, even on mobile platforms. To take advantage of multiple 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()*. *Isolate.spawn()* uses a top level function of an isolate to spawn that isolate. The top level function may reside in the same class or may belong to another class. *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 a source file in the local disk. To spawn an

---

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

isolate using *Isolate.spawnUri()*, the source file must have an entry point function named *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.

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

```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() {
```

```
25    ReceivePort receivePort = new ReceivePort();
26    SendPort sendPort;
         Isolate.spawnUri(Uri.parse("sample.dart"),null,receivePort.sendPort);
         // Spawns an isolate from sample.dart file
27
28    receivePort.listen((var message) {
29      if(message is SendPort){
30        sendPort = message;
31        sendPort.send(receivePort.sendPort);
32        sendPort.send("Hello");
33        sendPort.send(["a", "list", "datatype"]);
34      } else if(message is String) {
35        print("Reply: $message");
36      }
37    });
38  }
```

Difference from Actors

Although Dart isolates do not have shared state and use message-passing as the only means of communication between two isolates, isolates differ from actors [section 2.2] in many ways. The most significant difference is the principle behind spawning of an actor and spawning of an isolate. An actor should be very lightweight and cheap to spawn, but Dart isolates are resource heavy and slow to spawn. The implementation of actors found in languages like Erlang [section 2.3] and the 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 million [Incc]. A simple isolate in Dart consumes 5 to 7 MegaBytes[6] of memory, so the number of isolates per GigaByte of heap only reaches a few hundred. Based on these observations, it would be appropriate to say that the current[7] implementation of an isolate in Dart is — similar to a thread with properties like an actor.

Limitations of Isolates

Although Dart isolates follow the asynchronous message passing model which is suitable for distributed systems, it is not possible for an isolate in one Dart VM to send

---

[6]Based on the memory consumption of the two isolates from Listing 2.3
[7]Dart version 1.7.2

message to an isolate in another Dart VM. A message exchange between two isolates via SendPort/ReceivePort is possible only if the isolates are spawned locally in the same Dart virtual machine. This is a hindrance in making isolates distributed.

2.7 RabbitMQ - A Message Broker System

RabbitMQ is an open-source simple message broker software that implements the Advanced Message Queuing Protocol (AMQP). It serves as an intermediary for message passing between applications or components. It give applications a common platform to send and receive messages, and keeps the messages safe until intended subscribers receive them [Sofe].

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 applications that use 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'.

---

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

- MQTT
  Message Queue Telemetry Transport was developed by IBM. It provides lightweight publish-and-subscribe messaging, targeted for devices with limited resources and 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 sub protocol, RabbitMQ can transmit messages over HTTP [Soff].

  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 older browsers that do not have support for WebSockets, fallback mechanisms provided by SockJS[9] are used.

  JSON-RPC channel Plugin
  This plugin supports 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 messages are stored. It can buffer a large quantity of messages bounded only by the available resources of the machine.

RabbitMQ receives messages from a client via one of the protocols described in [section 2.7]. When sending a message, the client specifies the name of the queue 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 a request for dequeue, or only after acknowledging a previously dequeued message. This depends upon the parameters used during subscription to the queue.

If inflow of messages in a queue is faster than the outflow of the messages to its subscribers, both enqueuing as well as dequeuing becomes slower. Since messages

---

[9]SockJS is a JavaScript library that emulates WebSocket in browsers
[10]JSON-RPC is JSON encoded Remote Procedure Call protocol

are buffered into memory, as the quantity of messages increases the consumption of memory also increases. 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 RabbitMQ empties the queue as fast as possible to consumers. 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 hamper the performance of the whole application while setting it too high can cause 'out of memory' exceptions. Hence, based on the requirement and design of the consumer application, an appropriate prefetch-count should be determined.

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 numbers of consumers.
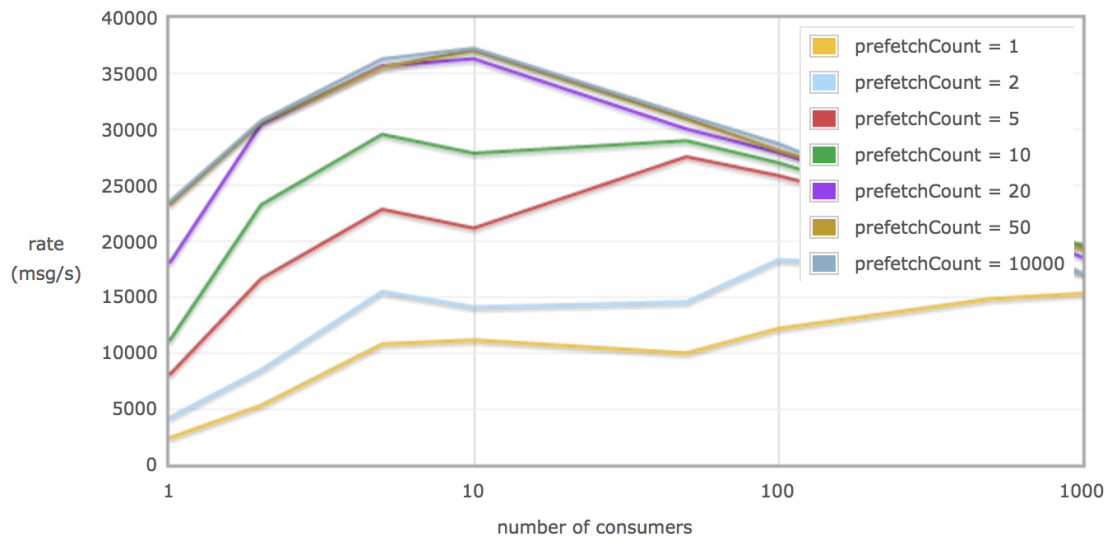

Figure 2.2: Chart[12]showing performance variation when prefetch count is changed in the consumer

[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 STOMP

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

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 as well [STO]. A STOMP client can either be a producer or consumer.

2.8.2 STOMP Library for Dart

Dart's 'pub'[13] is a public library sharing platform which allows users to share and reuse Dart code.

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[14] in Dart is available in Dart's 'pub' created by the 'Potix corporation'. It can perform most of the basic operations with a message broker system like connecting, creating queues, subscribing, enqueuing and dequeuing.

2.9 WebSockets

The WebSocket protocol enables two-way communication between client and server over a single TCP connection. It uses an origin-based security model, which is found in web browsers. It can be used for a 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].

WebSockets use two URI schemes: "ws://" for normal WebSocket connections and "wss://" for secured WebSocket connections [FM11].

---

[13]http://pub.dartlang.org
[14]https://github.com/rikulo/stomp

2.9.1 Security

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

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

2.9.2 Establishing a Connection

When establishing a WebSocket connection, an HTTP server receives a regular GET request with an offer to upgrade to a 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 DDE Framework, created as a result of this thesis, is designed to be distributed in nature with the concept of Actor Programming [section 2.2]. It provides an inherently distributed nature to applications built on top of it. The DDE framework itself is built using the concept of 'message-passing' [section 2.1] to alleviate any possibility of concurrency issues, making DDE applications thread-safe.

The framework is based on the following principles:

- All messages sent by the framework are based on the 'fire and forget' concept.

- The framework does not guarantee the delivery of messages.

- A message is delivered at most once.

- For simplicity, consistency and persistence, a message is always routed through a Message Queuing System [subsection 3.2.3], even though a target isolate may belong to the same isolate system in the same logical or physical node.

- Dequeuing a message (from a MQS) by a worker isolate is based on a pull mechanism, not a push mechanism.
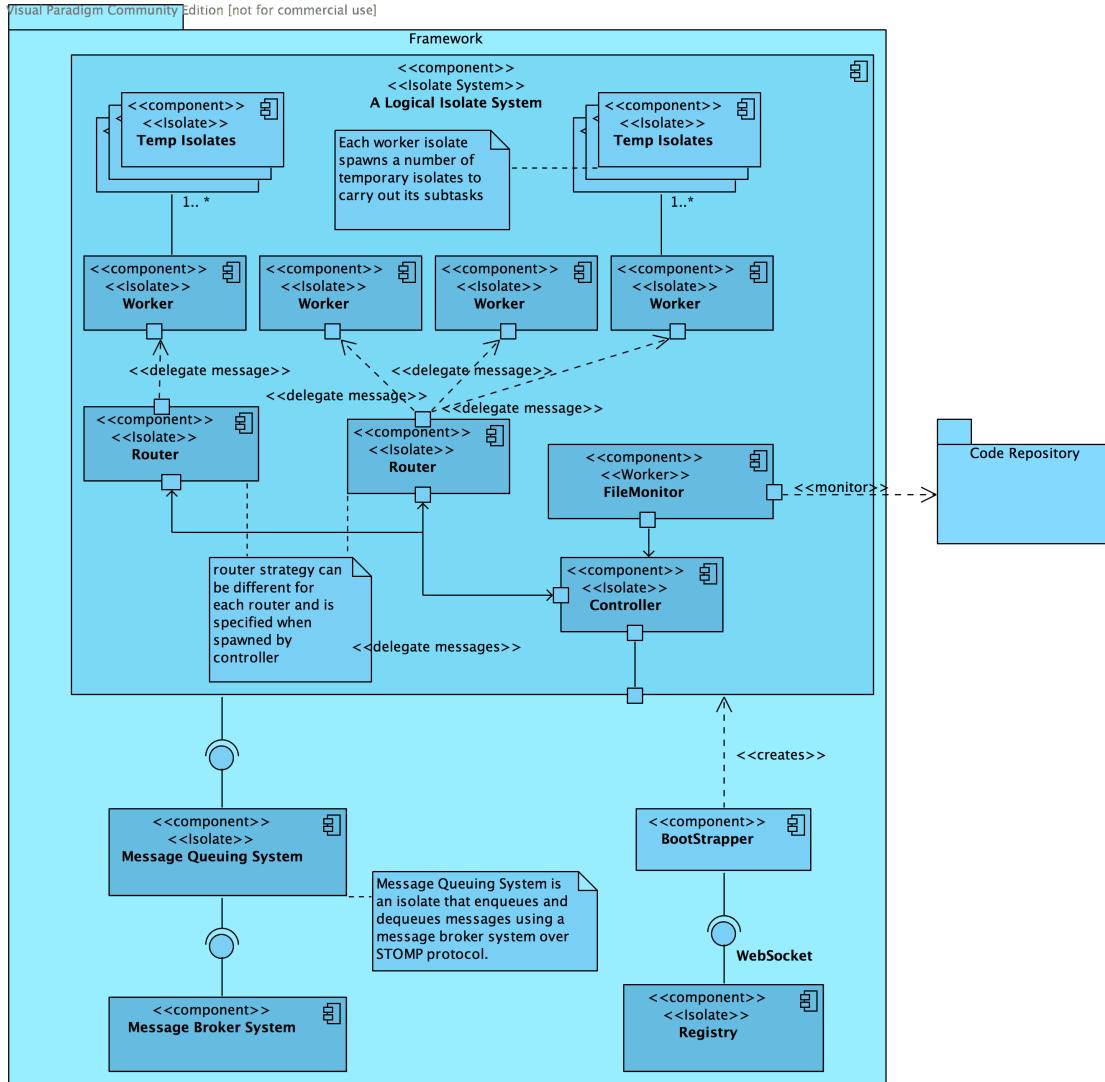
3.2 The Framework

Figure 3.1: Architecture of the framework

The DDE framework is comprised of an 'Isolate System', a 'Registry', a 'Message Queuing System', a 'Message Broker System' and an 'Activator'. Figure 3.1 depicts the major 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 consists of a group of 'Worker' isolates. It consists of different hierarchies which form an organization-like structure. The bottom most entities are 'Worker' isolates, which are managed by a 'Router'. Routers are managed by the 'Controller' and the controller is spawned by a top level isolate.

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

A 'Bootstrapper' in a physical node can start up several isolate systems. But, a logical isolate system is not limited to a single physical node. The 'Worker' isolates of an isolate system can be distributed across several systems.

Bootstrapping of an isolate system includes: generating a new unique id, 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 the 'Controller'. If a connection with the MQS fails or disconnects, the isolate system retries at a regular interval to establish a connection. Since the connection to the MQS takes place asynchronously, the isolate system moves forward and spawns the controller regardless of the establishment of connection to the MQS.

Adding Worker Isolates to an Isolate System

As an isolate system is a top level isolate, it spawns a controller. A controller spawns one or several routers and each router spawns worker isolates. Figure 3.2.1 shows a message flow sequence in different components while starting up an isolate system and deploying a worker isolate in it.
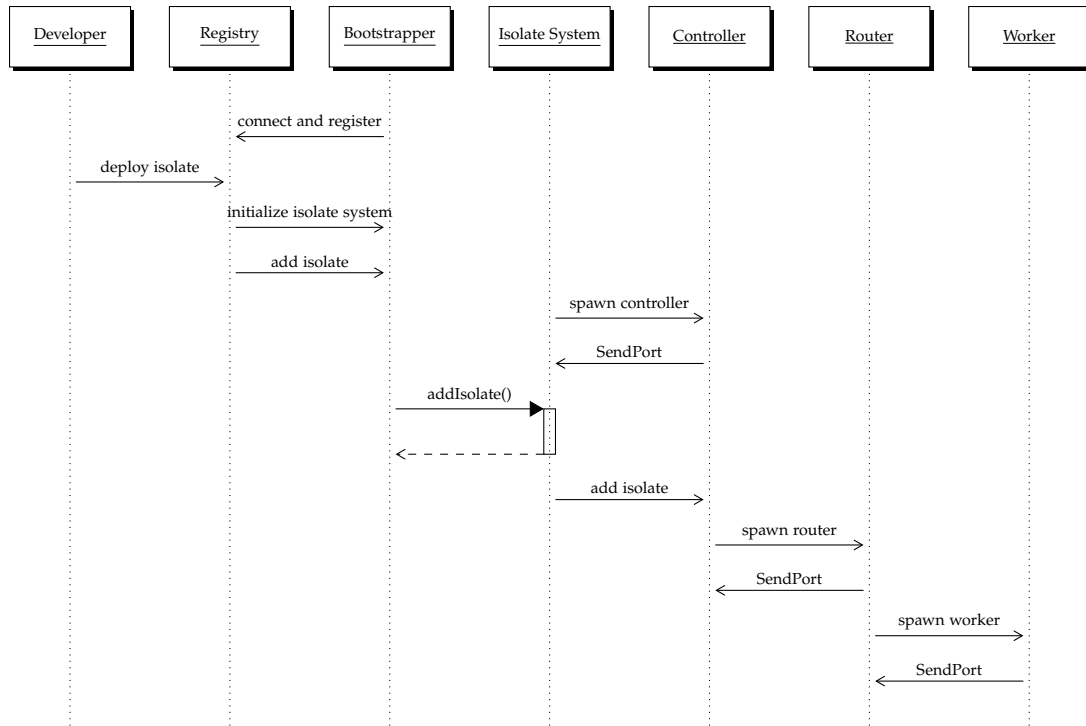
Figure 3.2: Adding a Worker isolate to an isolate system

An isolate system is initialized without any Workers running in it. The Workers are added with an appropriate load balancer after the isolate system is initialized. The *addIsolate* method starts up worker isolates in the isolate system. The method has the following required and optional arguments:

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

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

workersPaths – List of destinations where each of the isolates should be spawned. To spawn a worker isolate locally, 'localhost' should be used as the path, and to spawn in a remote node, a WebSocket path like "ws://192.168.2.9:42042/activator" should be used. The number 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 the local machine which is load balanced by the specified router.

routerType – Type of load balancing technique that one would like to use to effectively distribute incoming messages. By default, the DDE framework provides three types of routers: Round-Robin, Random and Broadcast. It is also possible to use a custom router implementation by extending the *Router* class of the framework. A custom router can be loaded by providing an absolute path (either file or http/https URI) for the source code location.

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

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

Message Handling in a Top Level Isolate

Typically, a message in an isolate system may arrive from three sources: the Message Queuing System via a WebSocket, the Controller via a ReceivePort or the Bootstrapper via direct method invocation. As Dart is a single threaded programming language [subsection 2.6.4], only one message is handled at a time by the top level isolate.

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

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

Messages arriving from the bootstrapper are either KILL messages or messages to get isolate details. A request to get detailed information of an isolate system and its isolates are made via web interface or RESTful web services of the registry [subsection 3.2.2]. When such requests are made, the registry forwards them to the respective bootstrapper [section 3.2.4]. The bootstrapper then invokes a function provided by the top level isolate. Then, the top level isolate sends a message to a controller to get the list of isolates the controller is managing and waits asynchronously for a future reply. As soon as a response is received from the controller, the top level isolate sends the response to the bootstrapper as a return value of the invoked function.

Another type of message is a 'KILL' message, which is used to terminate a worker isolate. Since a top level isolate does not directly manage running worker isolates, the message is forwarded to the controller which is next in the hierarchy. Similarly, when the 'KILL' message is sent to an isolate system to shut it down, the top level isolate forwards the KILL message to its controller and closes all open ports including WebSocket ports and ReceivePorts. After that it waits for the VM garbage collector to clean up the memory reserved by it.

Controller

Every isolate system has a single controller which is spawned by the top level isolate of an isolate system. A controller stays idle until it receives a message to spawn an isolate. After receiving a spawn message, the controller spawns and manages routers in an isolate system. It also spawns a 'FileMonitor' for each 'hot deployment' feature enabled router. When a RESTART message is received from a FileMonitor, the controller sends a RESTART_ALL message to the designated router, and the router restarts all worker isolates it has spawned.

A controller is also responsible for replying to the request for the list of isolates in an isolate system. It achieves this by keeping a detailed record of each router and number of worker isolates each router is handling. The list gets updated as soon as a worker isolate is killed or a new worker isolate is added.

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

Router

A router is spawned by a controller. The router creates and is responsible for a group of workers which are instances of the same class. 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 may be chosen at the time of deployment of a worker isolate.

A router uses a routing policy to distribute messages among the group of isolates it is handling. The default routing policies that are available in the framework are listed in 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 the *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 easily by selecting such workers using their deployment paths and returning them as a 'List'.

As a router manages the worker isolates it has spawned, it is responsible for effectively terminating and restarting the worker isolates. It also buffers messages that arrive when the worker isolates are not ready. This usually occurs during the creation and restarting of worker isolates.

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 dequeue request to the controller. This mechanism is present in the framework to prevent starvation of worker isolates. Starvation is caused when a dequeue message that may have been sent earlier does not reach the Message Queuing System due to a network issue or unavailability of the MQS.

Worker

The 'Worker' of the framework is an abstract class, which should be extended by the class that a developer of DDE applications creates. The worker unwraps messages that arrive from the router and retrieves the headers. 'sender' and 'replyTo' headers of messages are extracted before forwarding them to the worker instance. By unwrapping the messages that are encapsulated by various headers, the abstract Worker class makes sure that the messages are delivered to the target implementation of the Worker isolate immutated and in intended form.

To extend a 'Worker' isolate, one must implement the *onReceive* function which handles incoming messages and carries out application logic tasks. If a task is complex, a developer may divide it into subtasks and spawn temporary isolates to carry out those subtasks concurrently. Temporary isolates may be terminated once their subtask has been carried out.

send, *reply* and *ask* are helper functions provided by the abstract Worker class to send a message to another worker isolate. These functions automatically add the sender and receiver headers to messages that are sent.

Sending a Message To send a message from one worker isolate to another, the framework provides the *send* function. It requires the 'message' and 'address' of a target worker isolate as its arguments. A reply path can be optionally set, so that the replied message is sent to a different worker isolate for further processing. The named parameter[1] 'replyTo' should be used to set the address of the intended recipient of the reply 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 a message. In such case, the worker isolate specifically asks the target isolate to reply to this particular instance of the worker isolate.

A sample use case of 'ask' may 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

---

[1]Dart's named parameter is an optional argument in the function

isolates as messages, another instance of the same worker will not be able to respond to the request made in that connection.

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

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

Replying to a Message   To reply to a message the framework provides the *reply* function. It expects only a single 'message' argument because the response is sent to the worker isolate specified by the sender. The *reply* function can be used to respond to any message; eg:

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

Proxy

A 'Proxy' is a special type of a worker isolate. When a worker isolate should be spawned in a remote node, a router instead spawns a proxy isolate in a local node. Once the Proxy isolate is created, it connects to the 'Isolate Deployer' [section 3.2.4] of the remote node. After establishing a WebSocket connection with the Isolate Deployer, the proxy isolate forwards the request to spawn the worker isolate. After successfully spawning the worker isolate in the remote node, the proxy isolate forwards the messages that are sent to it by the spawner router. Each proxy worker maintains a separate WebSocket connection with an 'Isolate Deployer'.
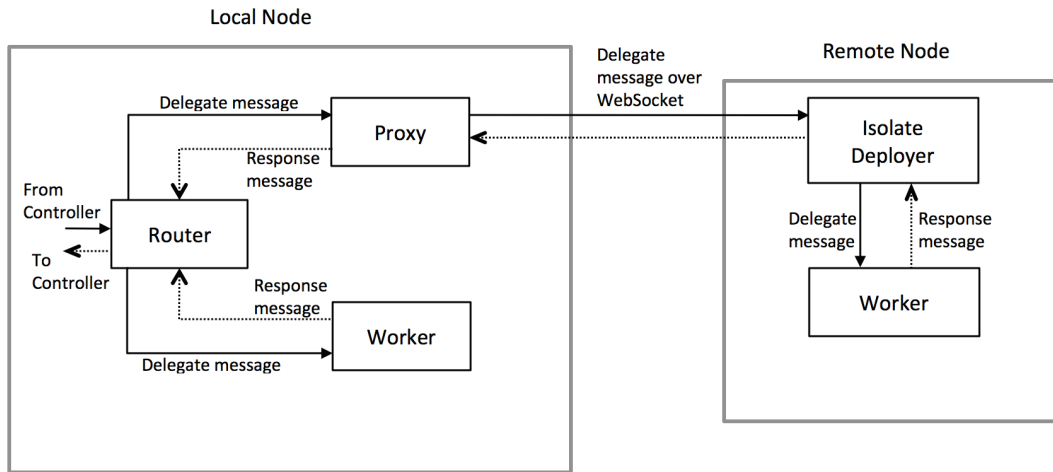
Figure 3.3: A Proxy Worker

FileMonitor

A 'FileMonitor' is spawned by the controller only if the 'Hot Deployement' flag for a worker isolate is set during deployment. The spawned 'FileMonitor' monitors the MD5 checksum of the source code from which the Worker isolate is spawned. If a change in the source file is detected, it sends a RESTART message to the controller. The controller forwards the message to the target router to restart the router's worker isolates.

3.2.2 The Registry

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

The basic tasks that a registry carries out are:

- Bootstrapping an isolate system, during runtime, in a local or remote node

- Providing a way to deploy, update or remove an isolate system

---

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

- Returning information about the deployed isolates by querying the individual isolate systems of a node.

RESTful API of the Registry

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

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

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:

```
1  [
2    {
3      "bootstrapperId": "266393094",
4      "ip": "54.77.239.244",
5      "port": "50189"
6    },
7    {
8      "bootstrapperId": "12133208",
9      "ip": "54.77.239.243",
10     "port": "50192"
11   }
12 ]
```

POST request to deploy an isolate system

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

Request Body:

```
1  {
2    "bootstrapperId" :"266393094",
3    "action": "action.addIsolate",
4    "messageQueuingSystemServer": "ws://54.77.239.200:42043/mqs",
5    "isolateSystemName" :"sampleSystem",
6    "isolateName" :"consumer",
7    "uri" :"http://54.77.239.221/sampleSystem/bin/Consumer.dart",
8    "workersPaths" :["localhost",
         "ws://54.77.239.243:42042/activator"],
9    "routerType" :"random",
10   "hotDeployment" :true
11 }
```

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:

```
1  {
2    "sampleSystem": [
3      {
4        "id": "sampleSystem/consumer",
5        "workerUri":
             "http://54.77.239.221/sampleSystem/bin/Consumer.dart",
6        "workersCount": 2,
7        "workersPaths": [
8          "localhost",
9          ws://54.77.239.243:42042/activator"
10       ],
11       "routerType": "random",
12       "hotDeployment": true
13     }
```

```
14    ]
15  }
```

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:

```
1  {
2    "bootstrapperId" :"266393094",
3    "isolateSystemName" :"sampleSystem",
4    "isolateName" :"consumer"
5  }
```

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:

```
1  {
2    "bootstrapperId" :"266393094",
3    "isolateSystemName" :"sampleSystem"
4  }
```

Response Status Code: 200 OK

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

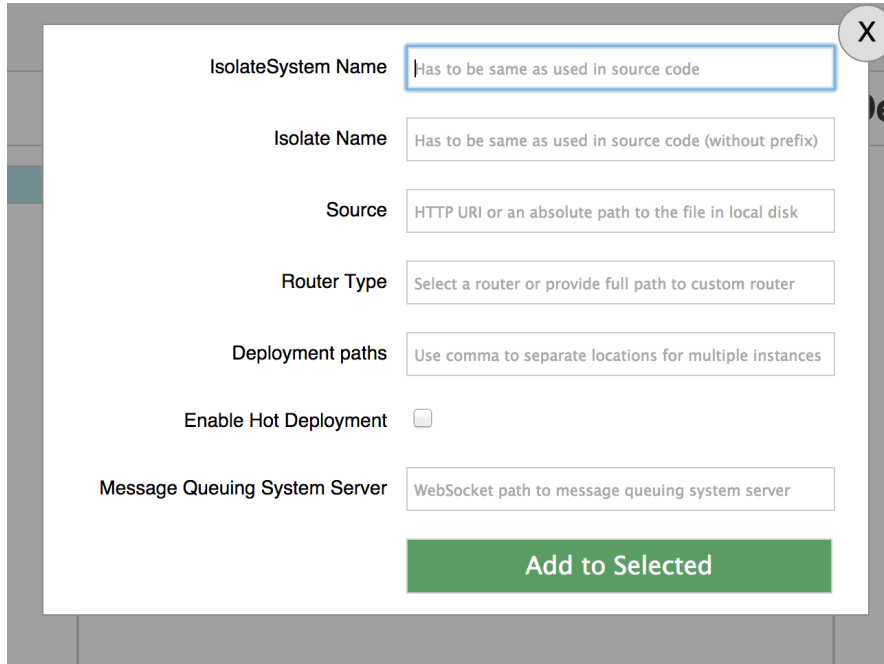**The Web Interface for the Registry**

Deployment of isolates can also be managed by using a web interface provided by the registry. The web interface must be started up as a separate process than the registry. It internally communicates with the registry using the REST API [3.2.2].

Figure 3.4 is a screenshot of the registry web interface showing a list of connected nodes and a list of deployed systems. The interface also shows the details of each

isolate. An individual isolate can be terminated using the 'kill' button next to an isolate, and an isolate system can be terminated using the 'shutdown' button next to the isolate system. To deploy a new worker isolate into a new isolate system, the '+' button may be used, which pops out a form shown in Figure 3.5. If the name of an existing isolate system is used when deploying an isolate, the isolate is added to the existing isolate system instead of creating a new isolate system.

Figure 3.4: Web interface of the Registry showing isolate systems and isolates in a node

Figure 3.5: Form of the web interface to deploy an isolate

3.2.3 Message Queuing System (MQS)

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

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

An MQS should be started up separately along with command line arguments to connect to a message broker system. The required command line arguments are: IP

address, port, username and password to connect to a broker. The 'prefetchCount' is an optional argument which defaults to 1, if not provided explicitly. A 'prefetchCount' is a Quality of Service header for the 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, the main isolate has to pipe all input/output data. In this case, an MQS is a top level isolate which handles all incoming and outgoing messages.

Enqueuer

An enqueuer of an MQS is a separate isolate. An MQS has only one enqueuer, which receives messages from the MQS and sends them to a message broker system (RabbitMQ [2.7]) using the STOMP [2.8] protocol.

Dequeuer

As opposed to an enqueuer, an MQS maintains a separate dequeuer for each queue of a broker. A queue also corresponds to each router running in an 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 using the STOMP [2.8] protocol. If a queue does not exist in the message broker system, the broker automatically creates it.

If a dequeuer is idle for too long, i.e. if the dequeuer isolate has not received any dequeue requests for a 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.

A dequeuer subscribes for messages from a message broker system with such options that new messages do not arrive to the subscriber unless previously dequeued messages have been acknowledged. This throttles the flow of messages from the message broker system and keeps itself and the isolates from being overwhelmed by a large number of messages and prevents 'out of memory' issues.

Messages in a dequeuer keep arriving as long as there are messages in a queue and they are being acknowledged. The buffered messages of a dequeuer stay unacknowledged unless they are flushed and sent out to the requesting isolate of an isolate system. As soon as a dequeuer acknowledges a message, it receives another message from its message broker.

---

[3]Dart version 1.7.2

[4]by default the timeout is 10 seconds

Multiple Instances of the MQS

It is possible for an application to have multiple message queuing systems for scaling up. If there are multiple identical isolate systems connected to different instances of an MQS, each MQS will have a dequeuer which subscribes to the same queue. However, a message is dispatched by the message broker system to only one of the dequeuers because it is distributed in a round robin fashion. This ensures that messages are fairly distributed among subscribers.

3.2.4 Activator

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

System Bootstrapper

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

A system bootstrapper is responsible for starting up an instance of an isolate system in a physical node. As it creates an isolate system, it can invoke methods provided by the top level isolate of the isolate system.

Isolate Deployer

An Isolate Deployer starts up a worker isolate in a node. The worker isolate is spawned without a local isolate system and as a part of an isolate system running in another node. This functionality expands an isolate system beyond a physical system. An isolate system can deploy a 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 a separate WebSocket channel with an isolate deployer.

3.3 Some Key Features

3.3.1 Hot Deployment of Isolates and Isolate Systems

The hot deployment feature of the DDE framework allows source code of an isolate to reside in a remote repository. For instance, source code of an isolate can reside in a git repository hosted in GitHub. When new code is committed to the repository, it gets immediately picked up by the application and the change is reflected without restarting.

After a node is bootstrapped, changes like adding, updating or removing isolates in an isolate system can take place. In such cases, isolates are killed and redeployed when they have finished processing and are 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 the 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. When 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 are restarted at runtime. This increases overall up-time of an application and keeps other components working even during modifications.

3.3.2 Migration of Isolates and Isolate Systems

Migration of isolates is the relocation of worker isolates or an isolate system during runtime. This requires killing one or more worker isolates in one node and spawning them in another node. The concept of hot deployment and migration brings enormous possibilities for a distributed system. Some of them are:

- Migration of isolates allows an application to adapt to deployment topologies in an easy way. This allows an application to respond to varying load and traffic.

- 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 imminent critical system failure, migration of worker isolates during runtime can make the application survive the failure.

3.3.3 Remote Isolates

The current isolate implementation in Dart[5] cannot communicate with other isolates over a network. Worker isolates in this framework have the ability to communicate with isolates that are running in a remote node. The communication underneath is taken care of by the framework so developers do not have to worry if an isolate is remotely or locally spawned. Two isolates running in different virtual machines can still belong to the same logical isolate system. Having remote isolates provides the DDE framework with clustering, high availability, greater scalability and isolate migration.

3.4 Typical Message Flow in the System

The DDE framework is based on the 'fire-and-forget' principle of message sending. Figure 3.6 shows a simple message flow while enqueuing a message and Figure 3.7 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 a ReceivePort.
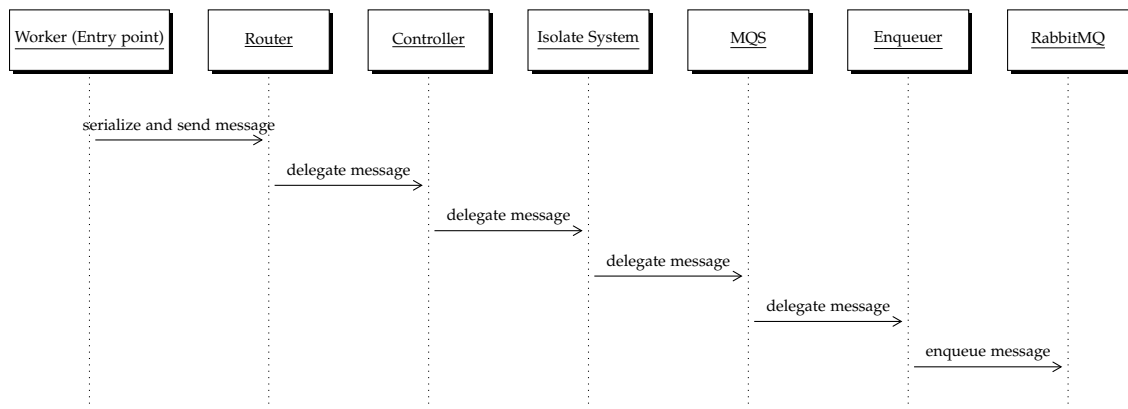

Figure 3.6: Enqueuing a message

---

[5]Dart version 1.7.2
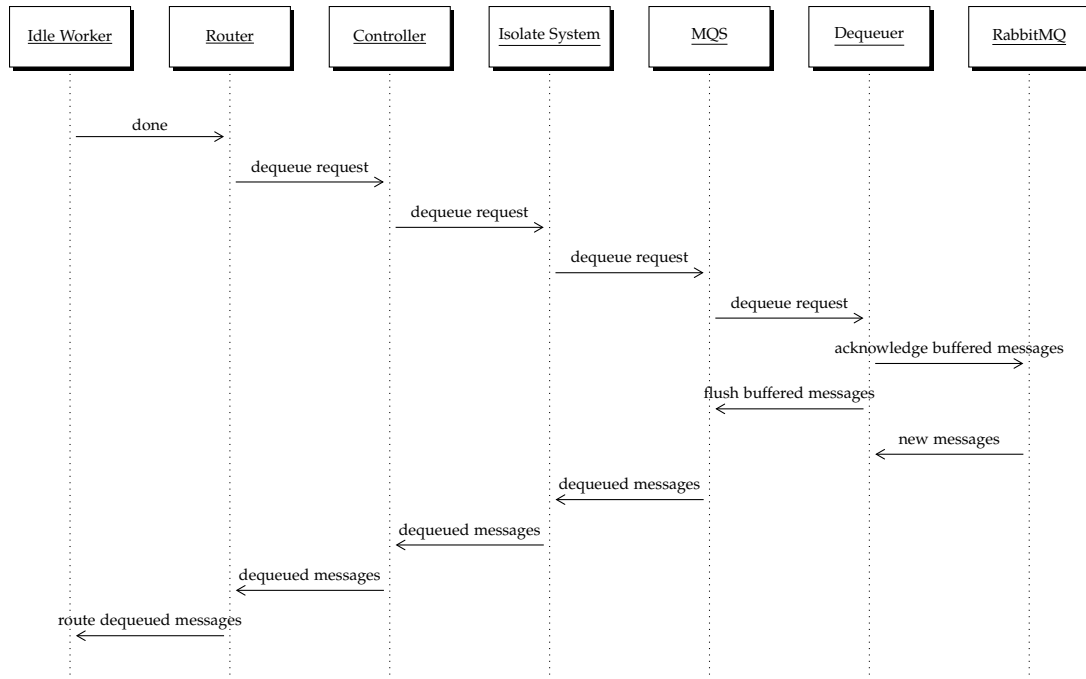
Figure 3.7: Dequeuing a message

3.4.1 Sample Message formats

A 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 the *send* function, the message is encapsulated with further information about the sender and receiver. The encapsulated message is forwarded to a spawner isolate, which is a router. The router again forwards it to a controller which again forwards to a top level isolate. Then the top level isolate adds another level of encapsulation and headers to the message so that an MQS knows the destination queue.

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

How *ask* works?
    The *ask* function has a subtle differences from the *send* function. The *ask* function should be used when the sender of a message expects a reply message. The abstract Worker class adds the full path of an isolate along with the unique id of the isolate when an 'ask' message is constructed. This is to make sure that the response from a target isolate reaches this particular instance of an isolate. When a router receives a message with a complete address (name along with its unique id) of a worker isolate, it routes the message to a specific worker regardless of its routing algorithm. If a worker isolate with the given unique id is not found in

the router's list of worker isolates, then the message is simply discarded. This is possible when isolates have been restarted or for some reason isolates were killed.

How *reply* works?

The *reply* function is simply a convenience for a developer. The *reply* function invokes the *send* function with the sender's address as the target isolate. If the message contains a 'replyTo' header then the message is sent to the address contained in 'replyTo' instead of the original sender. The *reply* function can be used for replying to messages for both the *send* and *ask* cases.

How 'KILL' works?

This is a special control message sent to isolates and isolate systems to shutdown themselves. If a KILL message is sent to a worker isolate, the message is enqueued to the end of its queue. Messages arriving after a KILL message are not sent to the worker isolate and are buffered until the isolate is restarted. The messages are delegated to a worker isolate only after its spawning is complete.

When a worker isolate finishes processing queued messages and encounters a KILL message, the isolate closes its ReceivePort[6] and stays idle. After sometime it gets garbage collected by the VM. Sometimes the garbage collector cannot clean up an isolate and a memory leak occurs. As a workaround, a custom 'Exception' is thrown deliberately by an isolate to terminate itself. The exception is thrown only after the isolate closes all ports. This workaround forcefully terminates the isolate and releases memory occupied by the isolate.

The abstract Worker class provides the *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, after issuing a KILL message, messages may continue coming from a controller to a router while restarting. 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' subsection 3.3.1 feature is enabled, if the source code of an isolate is modified and saved, each of the isolates that a router has spawned gets restarted. During which messages that arrive after a 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

---

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

of the registry or by sending a POST request to the registry. When a request to shutdown an isolate system is received, the isolate system closes all its ports including the ReceivePort, SendPort and the WebSocket connection with the MQS. After that a forceful shutdown is carried out by throwing a custom Exception. This is a workaround to free up the consumed memory 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 DDE framework at 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 the 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 brokers can be formed. This allows Message Queuing Systems to connect to the different members of a cluster.

Figure 3.8 is an example of a cluster formed by deploying a worker isolate 'A' across multiple isolate systems. The isolate systems are connected to separate instances of the MQS. Each MQS is connected to a separate node of the RabbitMQ cluster.
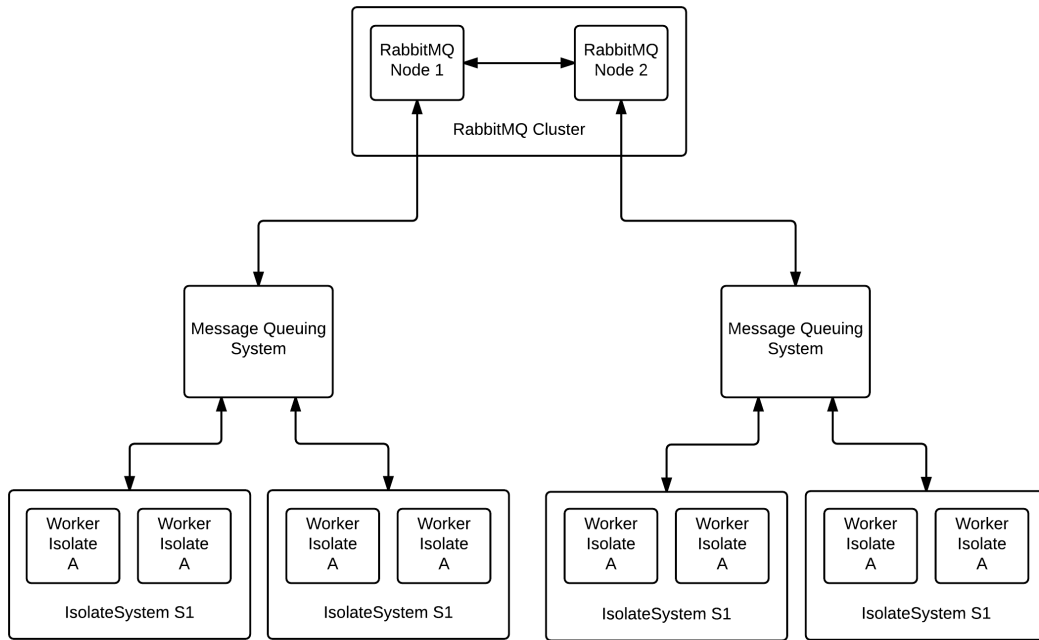
Figure 3.8: Forming a cluster in DDE Framework

3.5 Dart Libraries Used in the DDE Framework

Table 3.3 is the list of third-party libraries that are used by the DDE framework. All of these libraries are open source and available in Dart's pub.

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 DDE Framework

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

The example shown here prints a message if an '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 sender.

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

```dart
1  import 'dart:isolate';
2  import 'package:isolatesystem/worker/Worker.dart';
3
4  main(List<String> args, SendPort sendPort) {
5    Consumer printerIsolate = new Consumer(args, sendPort);
6  }
7
8  class Consumer extends Worker {
9    Consumer(List<String> args, SendPort sendPort) : super(args,
         sendPort);
10
11   @override
12   onReceive(message) {
13     switch(message['action']) {
14       case "print":
15         print("message['content']");
16         break;
17       case "send_back":
18         reply(message['content']);
19         break;
20     }
21     done();
22   }
23 }
```

4 Results

4.1 Applications Based on the DDE framework

To test the throughput of messages, scalability of the framework and round trip time of message delivery, two sample applications were created based on the DDE framework. The sample applications described in subsection 4.1.1 and subsection 4.1.2 were used for testing and evaluation of the framework.

4.1.1 Producer/Consumer Program

The producer-consumer program is a simple application. It has two worker isolates, a producer and a consumer. The 'producer', shown in Listing 4.1, starts producing 64-byte messages as soon as it is spawned. The 'producer' worker continues producing messages until it is terminated. The produced messages are targeted for the 'consumer' [Listing 4.2] worker which is done by setting the target address of the consumer to "mysystem/consumer" as the second argument of *send()* (line 31). Here, the first part of the address "mysystem" is the name of the isolate system where the 'consumer' worker is supposed to be deployed.

The 'consumer' worker shown in Listing 4.2 dequeues a message from its queue as soon as it is spawned. After receiving each message, the consumer prints it to standard output and invokes *done()* which sends a dequeue request to fetch another message.

Listing 4.1: Basic version of Producer Worker of Producer-Consumer application

```dart
import 'dart:io';
import 'dart:isolate';
import 'dart:async';
import 'dart:math' as Math;
import 'package:isolatesystem/worker/Worker.dart';

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

class Producer extends Worker {
  static const String consumerAddress = "mysystem/consumer";
  static const String Message64Bytes = "01234567012345670123456";

  StringBuffer data = new StringBuffer();

  Producer(List<String> args, SendPort sendPort) : super(args,
      sendPort) {
    description += "${args}";
    sendMsgWithDelay();
  }

  @override
  onReceive(message) {}

  sendMsgWithDelay() {
    while(true) {
      int timestamp = new DateTime.now().millisecondsSinceEpoch;
      Map message = {'createdAt': timestamp, 'message': Message64Bytes};
      int delay = 20 + new Math.Random().nextInt(480);
      sleep(new Duration(microseconds:delay));
      send(message, consumerAddress);
    }
  }
}
```

Listing 4.2: Basic version of Consumer Worker of Producer-Consumer application

```dart
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) {
    outText(message);
  }

  outText(var message) {
    print(message);
    done();
  }
}
```

4.1.2 Requester/Supplier Program

The request-reply program is a simple application. It has two worker isolates, a requester and a supplier. The 'requester' shown in Listing 4.3 sends a request message as soon as it is spawned. It randomly generates a request for a fruit, a number or a name. This request message is targeted for the 'supplier' [Listing 4.2] worker by setting its address as the second argument of *ask()*.

The 'supplier' worker shown in Listing 4.2 does a pattern matching on each request and based on the request message, it either replies with a random fruit, a random number or a random name to the original requester.

Listing 4.3: Requester Worker of Requester-Supplier application

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

main(List<String> args, SendPort sendPort) {
  new Requester(args, sendPort);
}

class Requester extends Worker {
  static const String SupplierAddress = "demosystem/supplier";
  DateTime startTime;

  Requester(List<String> args, SendPort sendPort):super(args, sendPort)
      {
    startTime = new DateTime.now();
    _sendRequest();
  }

  @override
  onReceive(message) {
    int receivedAt = new DateTime.now().millisecondsSinceEpoch;
    int requestedAt = message['requestedAt'];
    int repliedAt = message['repliedAt'];
    int rtt = receivedAt - requestedAt;
    print("Round trip time: $rtt");
    _sendRequest();
  }

  _sendRequest() {
    String requestMessage = "";
    int randomNumber = new Random().nextInt(3);
    if(randomNumber == 1) {
      requestMessage = AvailableSupplies.RANDOM_FRUIT;
    } else if (randomNumber == 2) {
      requestMessage = AvailableSupplies.RANDOM_NUMBER;
    } else {
```

```
37        requestMessage = AvailableSupplies.RANDOM_NAME;
38      }
39
40      Map message = {'requestedAt':new
             DateTime.now().millisecondsSinceEpoch,
             'requestMessage':requestMessage};
41      ask(message, SupplierAddress);
42    }
43  }
```

Listing 4.4: Requester Worker of Requester-Supplier application

```
1
2  import 'dart:isolate';
3  import 'dart:math';
4  import 'package:isolatesystem/worker/Worker.dart';
5  import 'AvailableSupplies.dart';
6
7  main(List<String> args, SendPort sendPort) {
8    new Supplier(args, sendPort);
9  }
10
11  class Supplier extends Worker {
12    Supplier(List<String> args, SendPort sendPort):super(args, sendPort);
13
14    @override
15    onReceive(message) {
16      String responseMessage = "";
17      switch(message['requestMessage']) {
18        case AvailableSupplies.RANDOM_FRUIT:
19          responseMessage = _randomFruit();
20          break;
21        case AvailableSupplies.RANDOM_NUMBER:
22          responseMessage = _randomNumber();
23          break;
24        case AvailableSupplies.RANDOM_NAME:
25          responseMessage = _randomName();
26          break;
27      }
```

```
28      Map response = {
29          'requestedAt': message['requestedAt'],
30          'repliedAt': new DateTime.now().millisecondsSinceEpoch,
31          'requestMessage': message['requestMessage'],
32          'responseMessage': responseMessage};
33      reply(response);
34      done();
35    }
36
37    String _randomFruit() {
38      int number = new Random().nextInt(5);
39      return ["APPLE", "ORANGE", "KIWI", "PEAR", "GRAPES"][number];
40    }
41
42    String _randomNumber() {
43      return new Random().nextInt(99999).toString();
44    }
45
46    String _randomName() {
47      int number = new Random().nextInt(5);
48      return ["Lorna", "Ambrose", "Domingo", "Kirsten",
              "Zachery"][number];
49    }
50  }
```

Listing 4.5: Supporting class that contains list of constants for pattern matching

```
1
2  class AvailableSupplies {
3    static const String RANDOM_FRUIT = "supply.fruit";
4    static const String RANDOM_NUMBER = "supply.number";
5    static const String RANDOM_NAME = "supply.name";
6  }
```

4.1.3 System Setup for Testing

The benchmarks of the sample applications were performed in Amazon EC2 [1] instances. A distributed environment with configurations shown in Table 4.1 was setup on multiple EC2 instances.

Table 4.1: 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

4.1.4 Observations From DDE Based Applications

Some observations made during the development and the deployment of applications based on the DDE framework were:

- The program source code based on the DDE framework is short, as evident from the source code of the applications listed in section 4.1.

- The worker isolate could not be run in a browser because of current limitations of the Dart VM.

- Setting up the system for the first time was complicated because it consisted of several components that needed to be started up separately.

- Once the system was setup, adding workers and isolate systems to the nodes was easy. Thus, deploying an application to a distributed system was easy.

- Shutting down a single isolate or an isolate system did not effect other components and nodes.

- The decoupling of isolate systems, the MQS and the registry allowed each component to start up and shutdown without severely affecting other systems.

---

[1] http://aws.amazon.com/ec2

- The worker isolates developed for an application were loosely coupled. They were coupled only in terms of messages.

- The REST API exposed by the registry simplified the deployment of isolate systems to different nodes.

- The Web interface of the Registry provided the ability to visualize and monitor the cluster.

- Messages sent to RabbitMQ were not lost during the restart of the system because they were persisted in RabbitMQ. The system continued to process messages after the restart.

4.2 Benchmarks

The benchmarks discussed here are the results of the evaluation of applications discussed in section 4.1. The results presented in this chapter were performed on Amazon EC2 servers[2] with configurations shown in table Table 4.1

4.2.1 Production Throughput of Isolates

The result shown in Figure 4.1 is the throughput of message production by the isolate system before the messages are sent to the MQS and broker for enqueuing. The production of a message in a worker isolate was throttled, so that there is no immediate 'out of memory' error from an overwhelming production of messages. Throughput was throttled by delaying the production of messages by a random amount of time ranging from 20 - 500 microseconds. The observations made here are the average throughput of each worker isolate as well as average throughput of all the producer nodes connecting to a single MQS instance.

As observed in the Figure 4.1, the total production rate of messages sharply increased when increasing the number of producers. In contrast, the average production of a single node was lower with a 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 one order of magnitude.
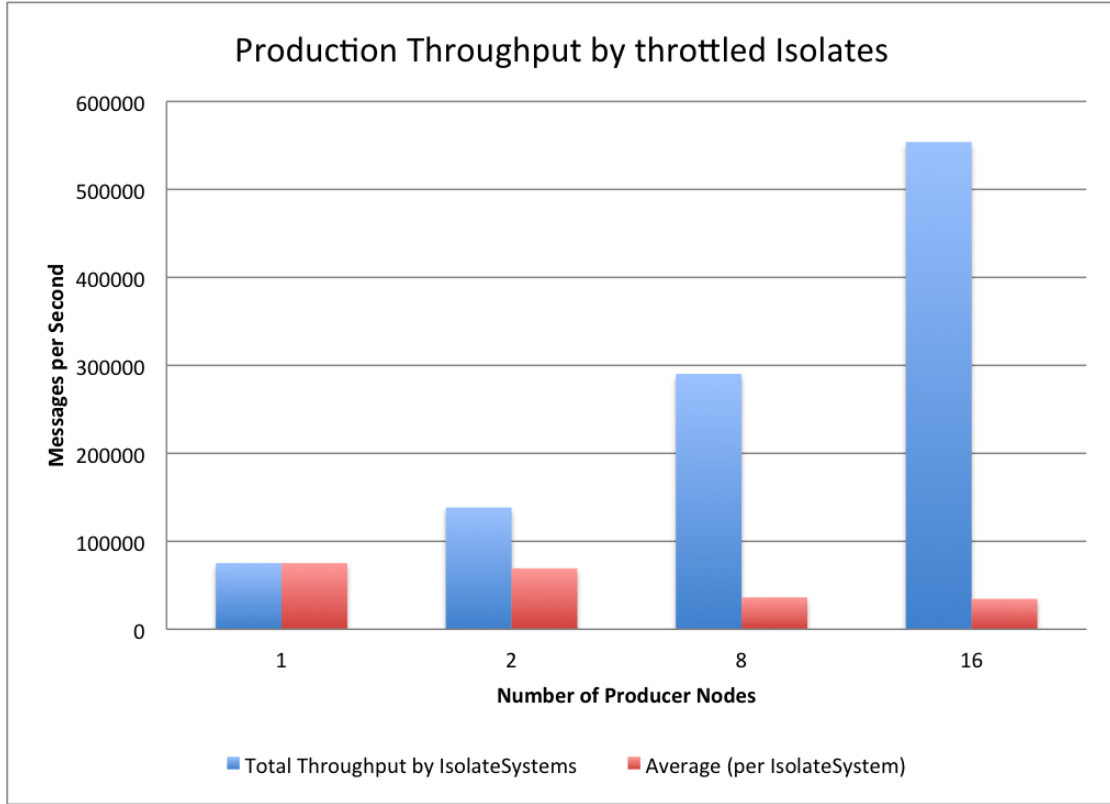
---

[2]EC2 servers of data center in Ireland

Figure 4.1: Production Throughput of Isolate

4.2.2 Message Size

The result shown in Figure 4.2 was obtained by having eight consumers with a prefetch count of 8 to dequeue existing messages from a queue.

The negative effect on message consumption throughput of larger messages is evident from Figure 4.2. The decrease with message throughput was more prominent between message sizes of 256 bytes and 512 bytes than that of between 64 bytes and 256 bytes.

Similar observation can be made from Figure 4.3 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.
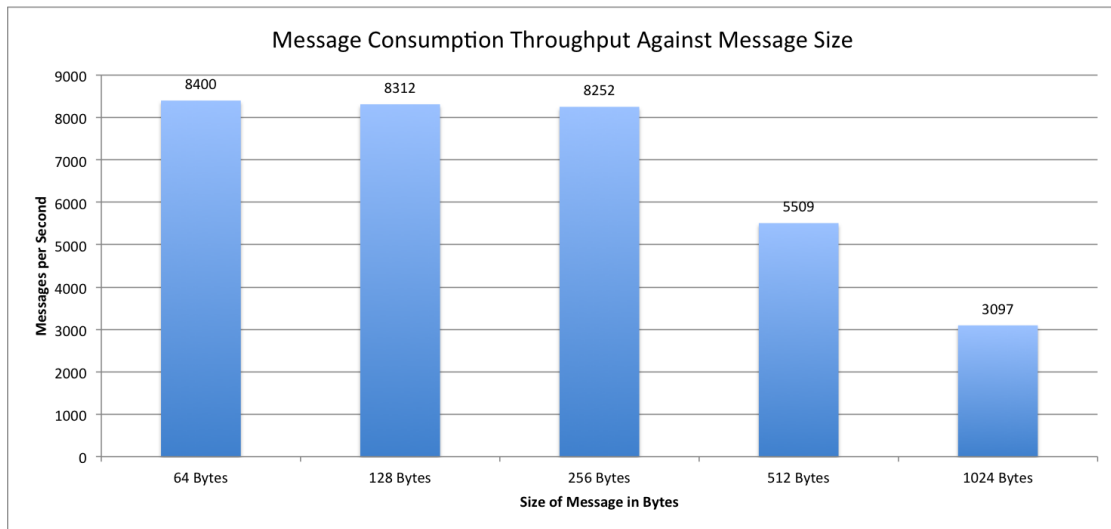
Figure 4.2: Message Consumption Throughput vs Message Size (Higher is better)
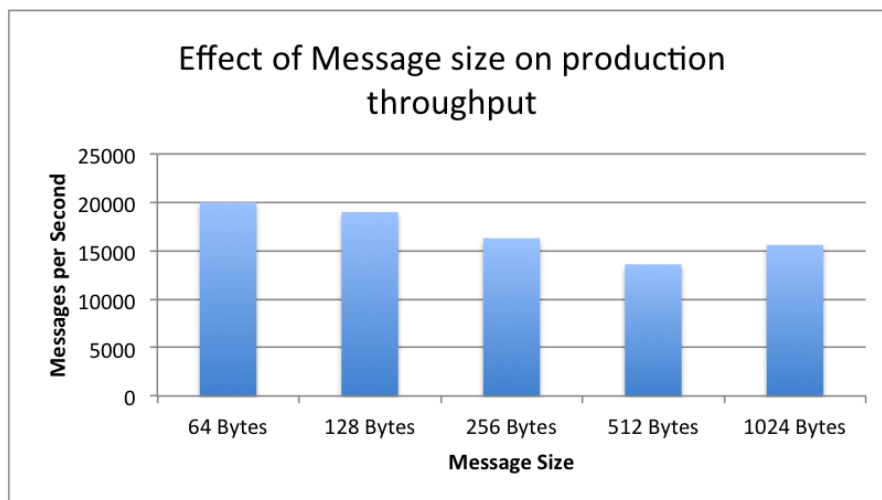

Figure 4.3: Message Production Throughput vs Message Size (Higher is better)

4.2.3 Number of Message Queuing Systems

Figure 4.4 is the result obtained by testing message consumption for 1 to 32 consumers distributed to one to four MQS instances connected to the same message broker. Adding MQS instances clearly increased message throughput. Nevertheless, adding more than eight consumers per MQS instance had a negative effect, as we can see from the decline

of throughput in the line charts for the one-instance case and the two-instance case. In the tests with one and two MQS instances, the optimum performance was seen when there were 8 consumers in total. But with four MQS instances, the throughput kept rising and supported up to 32 consumers without decline in performance. Nevertheless, the rate of performance increase was not as much as the rate seen when scaling up from two MQS instances to four MQS instances.

The positive impact of scaling out the MQS was seen not only on consumption throughput but also on production throughput. Figure 4.5 shows results of message production throughput by increasing the number of producers from one to eight, distributed across multiple MQS instances. The production rate measured here was the rate at which RabbitMQ enqueued messages, not the rate at which a worker isolate produced messages.
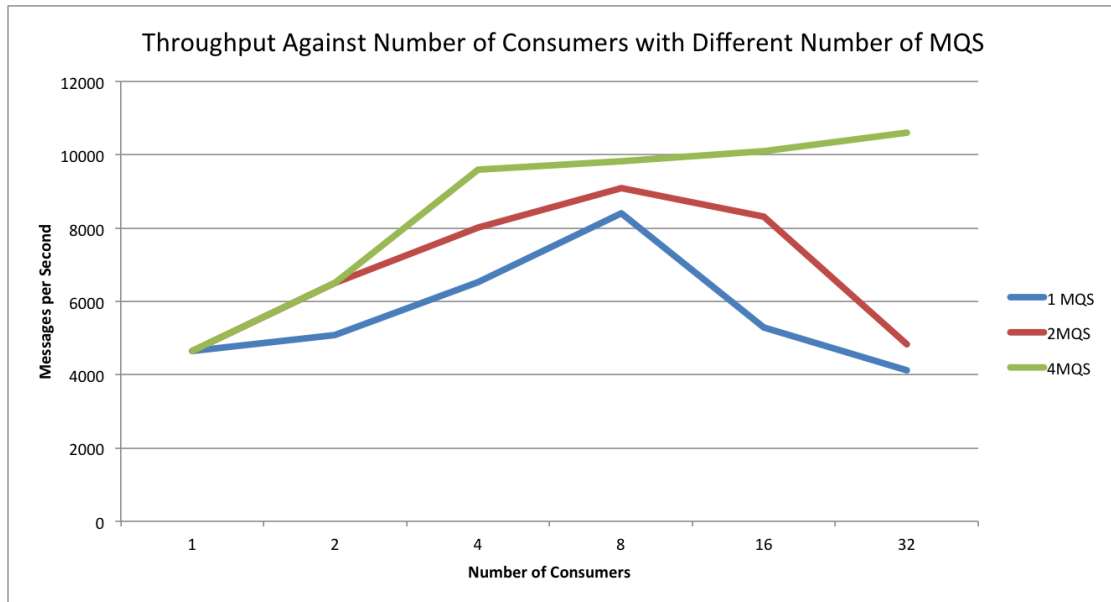

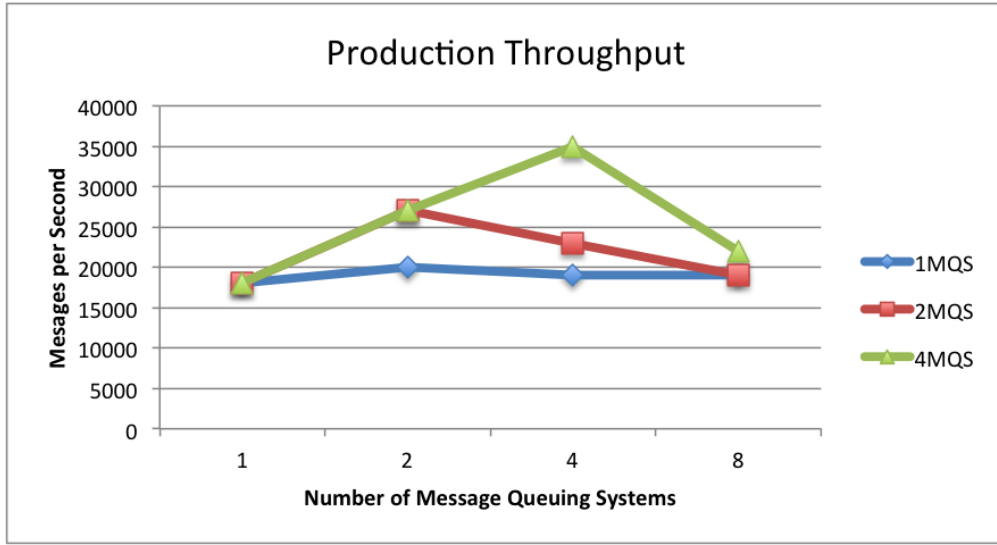Figure 4.4: Consumption Throughput on Scaled out Message Queuing System

Figure 4.5: Production Throughput on Scaling out Message Queuing System

4.2.4 Prefetch Count

Figure 4.6 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 for the prefetch-count [subsection 2.7.2] value set by the consumer while subscribing to the message broker system. Depending upon prefetch-count, increase, peak and decline of message consumption throughput can be seen for different numbers of consumers.

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

Increasing prefetch count had more positive impact in message throughput compared to increasing consumers. For instance, the increase in consumers from 1 to 16 resulted in the rise of approximately 2000 messages per second, while the increase in prefetch-count from 1 to 16 in the single consumer case resulted in an increase of message throughput by approximately 5500 messages per second.
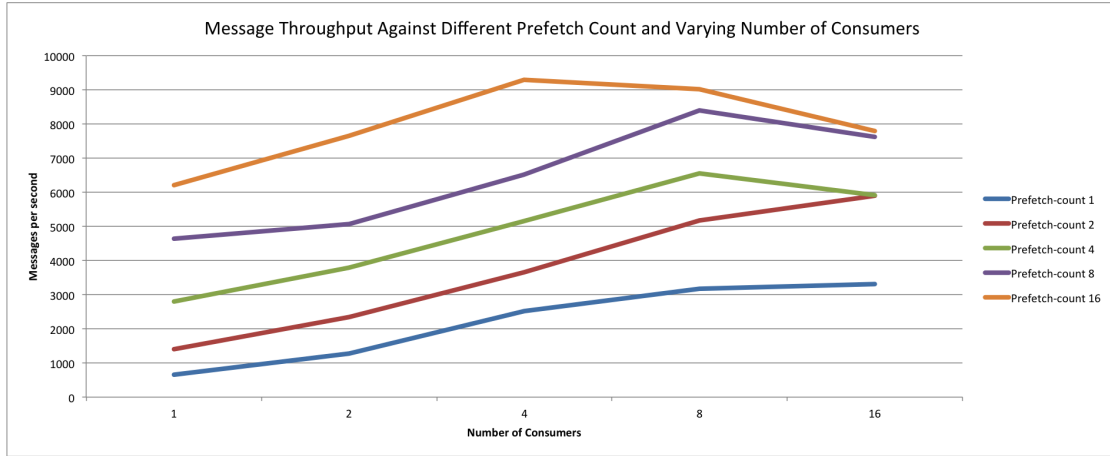
Figure 4.6: Message Throughput vs Number of consumers for varying prefetch-count (Higher is better)

4.2.5 Simultaneous Production and Consumption

In contrast to previous benchmarks, which were measured with either only producers or only consumers, the benchmarks in Figure 4.7 and Figure 4.8 show the consumption rate and production rate of messages when producers and consumers are run simultaneously in different nodes. Compared to what was seen in Figure 4.6 and Figure 4.4, the consumption rate is lower in this case. Nevertheless, the production rate of messages remained almost equal to that which was observed in Figure 4.4 with a single MQS instance.

If we compare Figure 4.7 and Figure 4.8, scaling out with producers and consumers connecting to different MQS instances had a significant positive impact in overall message throughput. Especially, in this case the consumption rate increased by almost an order of magnitude than that of using a single MQS instance. A slight increase in production throughput was also observed with additional MQS instances.
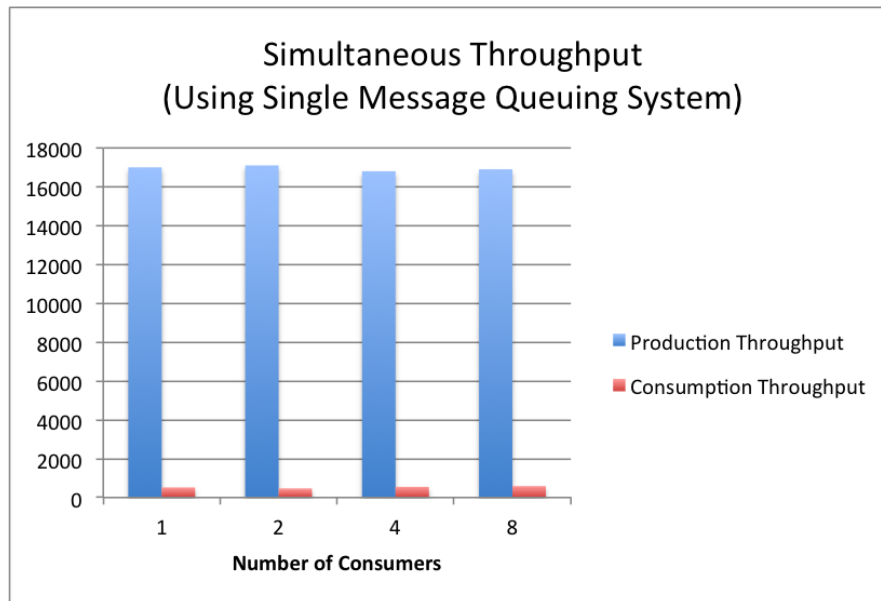
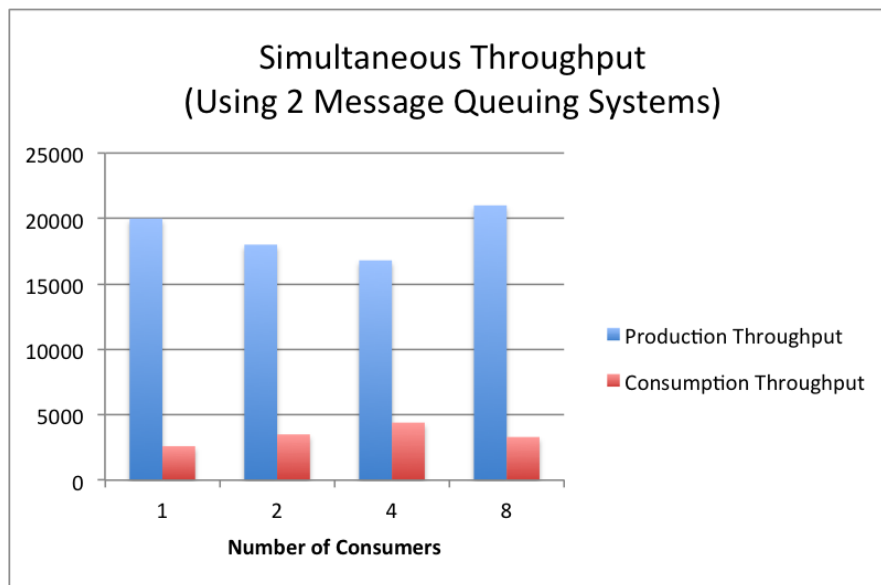Figure 4.7: Throughput during simultaneous execution in a single MQS instance

Figure 4.8: Throughput during simultaneous execution in two MQS instances

4.2.6 Throughput of Request-Reply

The request-reply application used for testing is different from the 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 a request for another message only after it receives the reply) whereas, in the producer-consumer application the producer creates messages regardless of the existence of consumers. Also, in the producer-consumer application any instance of a 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 reply message is sent to the worker isolate that originated the request.

For single execution of a message in the producer-consumer application, 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 must be enqueued and dequeued, then the reply message must be enqueued and dequeued. Thus, the steps required in the request-reply case are twice as much as in the producer-consumer case.

In Figure 4.9, when the number of consumers were increased we can see a linear increase in the production and consumption rate of messages. However, adding more suppliers had very little effect in overall throughput. The maximum increase in throughput by adding suppliers was seen when there were more consumers.
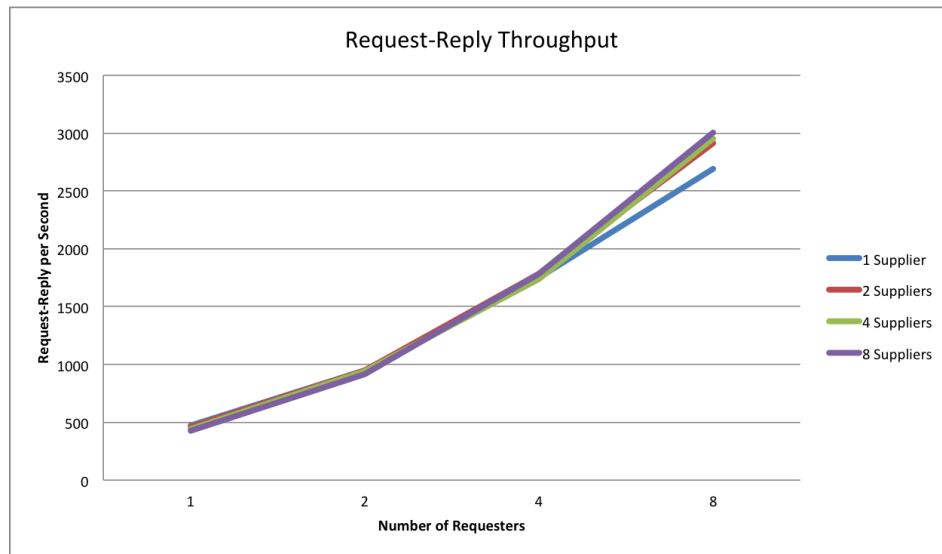

Figure 4.9: Throughput of request-reply

4.2.7 Round Trip Time in Request-Reply

The results seen in Figure 4.10 are the benchmarks of average round trip time (RTT) of messages in the 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 shown in Figure 4.10, we can see that there were inconsistencies in RTT of messages with respect to the number of suppliers and requesters. When there was only one supplier, the RTT of a message was usually higher compared to cases when there were more suppliers. Nevertheless, the least inconsistency was seen in the case of 8 suppliers and the least latency was observed in the case of a single supplier with a single consumer.
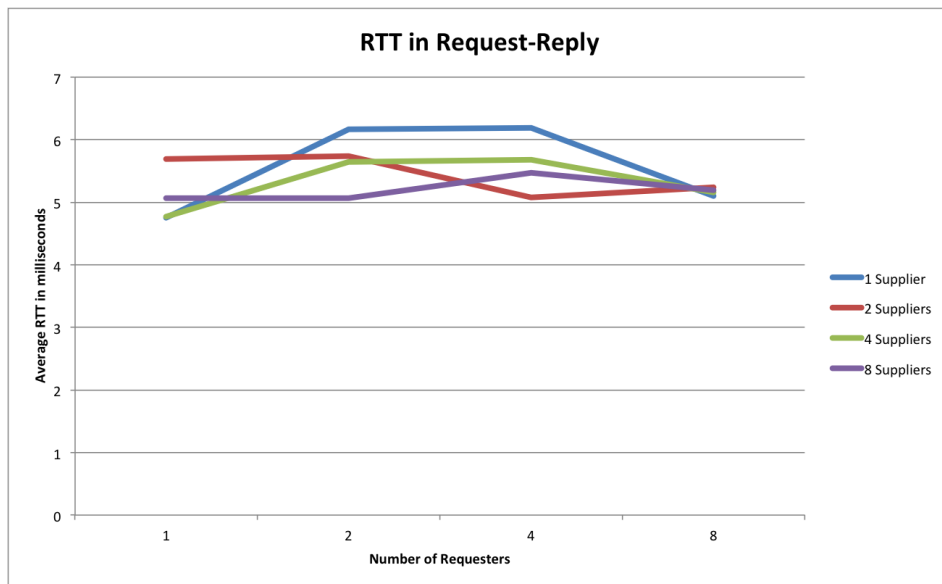

Figure 4.10: Round trip times of messages in request reply (Lower is better)

5 Discussions

5.1 Performance Findings

5.1.1 Comparision of Production Throughputs

As seen in Figure 4.1 and Figure 4.3, the difference seen between throughput at a worker isolate and throughput at RabbitMQ was quite significant. This is because the production of messages in an isolate were localized to one particular instance before the messages were transferred to the MQS over a WebSocket. Thus, it is natural the message production of isolates were high. Message production throughput increased with increasing the number of producers, but average throughput per isolate declined. One possible explanation for this behavior could be a bottleneck at the MQS to which the producers send messages. This explanation might seem strange because sending messages in an isolate are asynchronous, so it should not have been affected by an outside decoupled component. But, the reason behind this speculation is that since the MQS cannot accept messages at the rate they are produced, the messages get buffered in the internal queues of top level isolates. This takes up more heap memory reserved for the isolate causing slower execution. Similar effects probably occur in controller isolates, then in router isolates and ultimately in worker isolates. Hence, as the available heap memory of worker isolates becomes lower the production throughput decreases.

5.1.2 Effect of Message Size

From Figure 4.2 and Figure 4.3, we observed the negative impact of increase in message size. If we assess the amount by which the performance decreased, we can see that doubling the message size did not cause halving 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-byte messages and 1024-byte messages. The throughput of 64-byte messages was 8,400 messages per second whereas that of 1024-byte messages was 3,097 messages per second. If we assume that the 1024-byte message was the concatenation of sixteen 64-byte messages, then the overall throughput would become 49,552 messages per second, which is an overall improvement of about 6 times. This could mean that

for small messages, time is spent more on processing for persistence at the message broker rather than on Input/Output of messages.

A similar case for production throughput can be seen in Figure 4.3. With similar calculations for 64-byte messages and 1024-byte 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) for a message to maximize throughput might prove to be quite beneficial.

5.1.3 Effect of Scaling out the Message Queuing System

Scaling out the MQS clearly had positive impact on overall throughput. Higher number of MQS instances supported more consumers with good overall performance. The decline of performance seen in tests with 1 MQS instance and 2 MQS instances were probably because of the behavior of RabbitMQ when too many consumers are connected as we discuss in subsection 5.1.4. But, the test seen with four MQS instances suggests otherwise. With four MQS instances, 32 consumers were supported without a drop in performance. From which, we can speculate that the MQS may be causing the limit in throughput, as the MQS has to distribute messages to the connected systems. As more consumers are connected to the MQS, more CPU time and memory are required.

5.1.4 Effect of Prefetch Count

In the observations made in Figure 4.6, throughput scaled almost linearly to 8 consumers for every prefetch-count (except when the prefetch-count was 16). Adding more consumers after 8 resulted in a decrease of throughput, which is quite similar to the RabbitMQ benchmark [Figure 2.2] presented in subsection 2.7.2. In the benchmark of RabbitMQ, a decline of throughput was seen with 8 to 10 consumers.

The greater the number of consumers, the more work is required by RabbitMQ to keep track of them. Hence, it could be the same reason we saw the decline in performance with more consumers in Figure 4.6.

In a distributed system, a sudden performance change could be due to a number of factors, such as network speed, the system in which the isolate system is running or because of a bottleneck in the MQS. 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 4.6 suggests that the reason could be due to the broker handling too many consumers for a queue.

5.1.5 Effect of Simultaneous Production and Consumption

In Figure 4.7, we can see the negative impact of starting producers along with consumers when the producers were producing messages continuously. Again, when the producers and consumers were split to connect to separate instances of the MQS, the performance improvement was more prominent in message consumption.

By comparing Figure 4.7 and Figure 4.8, the culprit of this behavior seems to be the MQS. Even though the MQS has separate isolates for enqueuing and dequeuing, it has only one top level isolate [subsection 3.2.3]. Obviously when there is a large quantity of messages from a producer in queue, the dequeue requests that arrive from the consumer and the messages that arrive from the Dequeuer go further back in the internal isolate queue. The improvement of consumption throughput seen in Figure 4.8 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 the dequeuer get processed much quicker compared to the previous case.

5.1.6 Throughput of Request-Reply

The benchmark of request-reply [subsection 4.2.6] showed almost linear improvement with increase in the number of requesters. But, increasing the 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 the *ask* method to send messages. 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 suppliers, but this is quite minimal and could have been affected by other factors like RabbitMQ performance, change in network latency, etc. There is not enough data and evidence to support improvements in throughput with more suppliers. Testing with more consumers till the supplier saturates would yield a better understanding of what affects throughput in the request-reply case.

5.1.7 Round Trip Time of Request-Reply

The observation seen in Figure 4.10 was inconsistent. It is difficult to make any strong conclusions from the data. Nevertheless, it seems that the increase in number of suppliers decreases latency, as seen in the case of eight suppliers against a single supplier.

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

5.2 The DDE Framework

According to the observations made after creating and testing the applications based on the framework (as discussed in subsection 4.1.4), we can infer the applications responded well when the system was scaled up. We can see a rise in message throughput when additional isolate systems are added. The rise in message throughput was even more when the number of MQS instances were increased.

During testing, restarting the message broker system (RabbitMQ) and restarting the message queuing systems did not result in failure of the whole system. The system operated normally as soon as they were back online. Thus, we can say that the system was highly available and it is suitable for systems that must be "run forever".

5.3 Problems/Issues

5.3.1 Dart Induced Issues

Even though Dart is a fairly mature language, it still has some bugs and incomplete features. Some of the issues could be solved by workarounds while others could not be easily resolved. All issues listed here were faced in Dart version 1.7.2.

Dart's documentation for isolate [Goo] mentions that there is experimental support for the following methods in isolates: *addErrorListener*, *addOnExitListener*, *kill*, *pause*, *ping*, *removeErrorListener*, *removeOnExitListener* and *resume*. But while developing the framework, it was found that these methods were not implemented. The implementation of these functionalities would have particularly helped in the implementation of the supervision strategy. Moreover, the implementation of features like 'hot deployment', 'migration' and isolate termination would have been cleaner and simpler compared to the current workarounds present in the DDE framework.

We have already discussed the workaround implementations for *kill* in subsection 3.4.2 and *ping* in section 3.2.1.

Another issue found in the Dart VM for browsers which made it impossible for an isolate system designed in the DDE framework to run was the lack of support for multi-level isolates. The 'Dartium' browser [1], only supports spawning of isolates up to one level deep. i.e. Spawning of an isolate by the spawned isolate was not allowed. The reason behind the lack of this implementation is not clear.

---

[1]a variant of Chromium browser with built in dart VM

5.3.2 STOMP Library for Dart

As mentioned in subsection 2.8.2, a third-party library for the STOMP client in Dart was used from Dart's 'pub'. Although it had good support for STOMP 1.2, it was missing a feature to set the prefetch-count [subsection 2.7.2] when subscribing as a consumer. Without setting the prefetch-count the system became unusable, especially when there were many messages in the queue.

As the project was hosted in GitHub[2] and is licensed under Apache 2.0[3], I forked[4] the repository and added the functionality so that the STOMP client could support setting the prefetch-count. After making modifications and testing, I created a pull-request[5], which was swiftly merged[6] by the original developer and was updated in Dart's pub as a new version.

---

[2] `http://github.com`

[3] `https://github.com/rikulo/stomp/blob/master/LICENSE`

[4] make copy of the source code and add custom modifications

[5] request to merge my changes into the master branch of the original repository

[6] `https://github.com/rikulo/stomp/pull/15`

6 Conclusion

The DDE framework, created as a result of this thesis, shows that isolates of the Dart language do not have to be limited to a single virtual machine.

The DDE framework provides structure for developers to create applications based on the actor programming model in the Dart language. This framework uses the existing actor-like nature of isolates and provides easy to use functions to make message sending similar to the actor programming model.

As a result of using this framework, applications become easily scalable and offer higher availability with virtually no down time during code updates. Scaling up can be performed in several ways: deploying more isolates in an isolate system, replicating isolate systems across multiple machines, increasing the number of message queuing systems, forming a cluster of message broker systems (RabbitMQ) or the combination of these.

Moreover, the DDE framework offers developers the capabilities to monitor code updates and restart worker isolates automatically. The web interface and REST API available in the 'Registry' allow developers to visualize, easily deploy, terminate individual or groups of isolates, and even shutdown isolate systems.

The message broker system (RabbitMQ) of the DDE framework allows applications to be loosely coupled and improves the overall fault-tolerance of the system. The clustering ability and message persistence provided by RabbitMQ allows messages to be saved in several machines ensuring message durability.

Nevertheless, the DDE framework has not yet implemented some features required by certain users. For example, it does not have a security implementation of its own.

The preliminary tests and benchmarks of the DDE framework are encouraging. For the proof of concept implementation, the throughput of message production and consumption were good. With further profiling and optimization, throughput and latency improvements can be achieved.

7 Future Directions

The work done in the DDE framework in this thesis may be extended and improved in a number of directions. First, the most important missing feature, security, requires a concept and implementation. The DDE framework heavily relies on the underlying transport layer to provide security, additional application level security would provide the DDE framework more safety.

Another important feature is to support developers in testability of DDE applications. This would improve programmability and help developers to write quality code.

The implementation of the supervision strategy is far from perfect in the presented DDE framework. The current implementation of the supervision strategy does not properly follow the "Let it Crash" [subsection 2.3.1] philosophy. As discussed in subsection 5.3.1, the implementation of this feature would have been simpler and cleaner had Dart supported all of the unimplemented features of isolates. Nevertheless, the Dart language is evolving quickly and those features may eventually be implemented. Even if those features do not make into the major version of Dart any time soon, it is possible to make implementations using workarounds.

Another interesting idea would be the introduction of adaptive load balancing and automatic migration of isolates based on different heuristics like load on a system, data locality or a user's geographic location.

One feature to improve the fault-tolerance of the system would be to change the implementation for FileMonitor [section 3.2.1]. The current implementation of the FileMonitor simply monitors the file from a remote location and periodically fetches it to calculate the checksum. In case of modification in the file, the FileMonitor informs the controller which instructs a router to re-spawn the worker regardless of the contents of the source. This means there is a vulnerability to deploy incorrect code which could result in failing to start up the isolate. The ability to rollback to a previous working version of the source in case of failure of spawning would improve the resilience of the framework.

Improvements in the user experience of the registry [section 3.2.2] web interface would add more value to the usability of the framework.

Compared to Akka actors, Dart isolates require more memory and more time to spawn. Creating an alternative lightweight entity to isolate, but having similar properties of an isolate would make the applications more lightweight, less resource hungry

and reduce the cost of dynamic isolate creation.

An improvement to add flexibility to the framework would be implementing an internal message broker in the framework itself, which may not provide message persistence as a broker does. The option to choose the internal broker or an external broker would provide more flexibility for applications and developers. This could be an added attraction to developers who need high throughput in their applications. For instance, if an application demands high throughput but does not require message persistence, then the internal queue instead of external message broker can be chosen.

The ideas discussed in this chapter are only a few of many other improvements that can be made in the framework.

Acronyms

DDE Dart Dart Everywhere.

JSON JavaScript Object Notation.

MBS Message Broker System.

MQS Message Queuing System.

RTT Round Trip Time.

URI Uniform Resource Identifier.

UUID Universally Unique Identifier.

VM Virtual Machine.

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 Java Documentation.* `http://doc.akka.io/docs/akka/2.3.7/AkkaJava.pdf`. Last Accessed: 2014-11-19.

[Incc]   T. Inc. *Akka Toolkit.* `http://akka.io`. 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).* `http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-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`.