Aldebaran
A High Performance Distributed Processing Environment

Markus Fix

2015-06-26

Compiled on 2015-06-30 at 16:47:00

Contents

List of Figures

1 Preamble

This document serves as a record of an iterative design process. It is *not* a complete specifiation of the system we need to build. We cycle through multiple iterations as we refine our understanding of the challenge and contemplate possible designs.

2 The Challenge

2.1 Design a streaming data processing system

We are asking you to design a streaming data processing system. You have as much time as you want to complete the exercise. Final product should include some kind of design document of the individual components in the system and how they interact. Where you feel it necessary, feel free to supply code to illustrate your point. If you are designing for the use of specific technologies, algorithms etc, please be sure to indicate. Please address as part of your final work any tradeoffs you made between requirements and which ones you ended up favoring.

2.2 Application

Our application consumes messages representing financial trade data. Requests to purchase and purchases. A request to purchase can be broken down into one or more purchases. In an attempt to get the best price, the total number of shares purchased can exceed the number requested. Should the number of shares purchase exceed the requested amount an alert will be generated.

2.3 Example messages

Request to purchase:

```
id:1
# shares: 100
client: Sendence
stock: AAPL
```

Purchase:

```
id:2
initial order id: 1
# shares: 90
client: Sendence
stock: AAPL
```

Purchase:

```
id:3
initial order id: 1
# shares: 30
client: Sendence
stock: AAPL
```

When both purchases 2 and 3 are processed, the number of shares purchased will exceed the total amount requested in the initial order and an alert should be generated. Any subsequent purchase shouldn't cause any additional alerts.

2.4 System Requirements

- 250,000 msg/sec throughput (with ability to go higher)
- 5 milliseconds message processing latency
- System should be able to continue processing in the face of failure in any individual component
- Should be usable in both a private datacenter and AWS type environments
- Alerts should be received by final intended recipient only once
- While this exercise is for a specific to a given use case, the general core of the system should be reusable. When the next use case comes along, we don't want to reimplement all the basics over again.

2.5 Additional details

- Source of messages: You can use any message bus you want as entry point of data into your system so long as it is available as an open source product (Kafka, RabbitMQ, ActiveMQ). If none of the existing messaging systems meet the needs of the system, please describe at a high level the qualities needed for your message bus and roughly how you might want to implement it.
- Alerts will be pushed out of our system to any number of downstream systems, another message bus, to an external alerting/monitoring system such as Nagios, an SMS messaging provider such as Twilio, as an iPhone Push Notification or an email. It is the responsibility of our system to assure only one alert is sent without the ability to coordinate with the system that provides actual means of delivery.
- Cost of implementation is a factor. A system that costs $1 million a year to run might be worth some tradeoffs with requirements when compared to a system that costs $10 million a year to run.

2.6 Example topics of conversation

- What message processing guarantees does the system provide?
- What are the failure characteristics?
- What are the recovery characteristics?
- What could cause latency issues?
- What could cause throughput issues?
- Are there any single points of failure?
- What metrics within this system do you consider to be important and at what points would you want to capture them?
- What are the monitoring options for each component you've specified? e.g. SNMP, JMX, direct notifications
- If you're debating between options for any component in the system, what are the pros and cons of each and how would you make a final determination? e.g. Kafka vs RabbitMQ

3 Design Process

3.1 Iteration 0

We considered if RabbitMq, would scale well enough for the demanded throughput (250k msg/sec). A quick search found a study done by Pivotal where RabbitMq was scaled to deal with 1 million messages per second. Finding this we shelved the questions regarding message bus technologies until later.
Let's start with a naive model of a system that processes the stream of purchase requests and purchases. We will ignore questions of scaling, resilience and coordination for now and return to them later.
Figure 3.1 illustrates the basic message sequence. We receive a trade data from the TDS into a queue. Our Auditor consumes trade data and watches the limits for each purchase request. If the sum of purchases exceeds the defined limit the Auditor publishes an alert to the Alert Queue. The Fire Lookout consumes alerts from the Alert Queue and distributes these form to the registered Alert Channels. Each Alert Channel has it's own Alert Channel Queue which is processed by a dedicated Channel Boy. The Channel Boy consumes from the queue and notifies the Alert Channel exactly once for each alert.

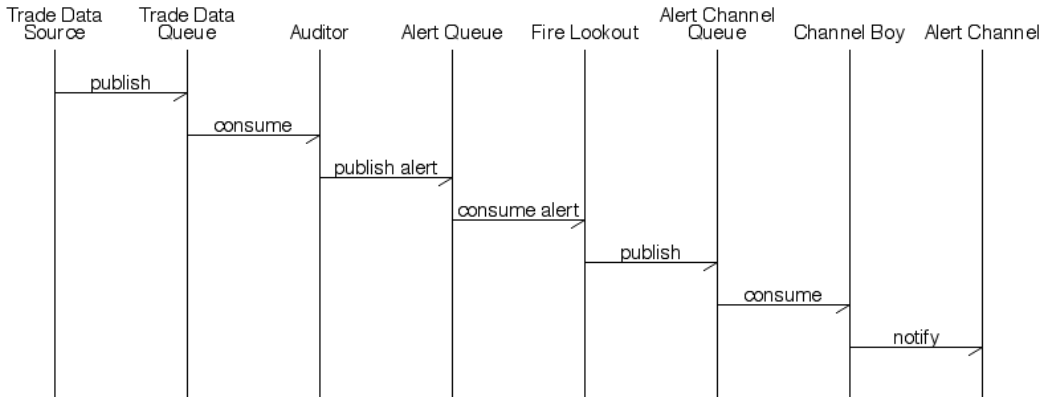

Figure 3.1: Naive Message Flow

3.1.1 Assumptions

Some details are underspecified in the challenge description. We've made the following assumptions:

1. Events are semi ordered sequences. A purchase request event always arrives before the related purchase events.
2. We can close an open book (awaiting purchase events) after a defined delta t or right after we have generated an alert. Once closed the book can be garbage collected.
3. We do not need to keep a persistent record of books or alerts.

3.1.2 Questions

The Auditor will need to keep a book open for each purchase request to accumulate purchases.

1. How will we manage these books in a distributed system?
2. How can we minimize or even eliminate coordination if we run multiple Auditors to scale up performance?
3. Can we use routing keys to tag each purchase and then route it to the correct book for accumulation?

3.2 Iteration 1

We need to refine the Auditor component. Here's what we know so far:

1. Each new purchase request event (PR) will open a new book where we accumulate purchases.
2. Each purchase event (P) will be consumed by the open book and accumulated.
3. If after accumulating a P the sum of all shares purchased exceeds our defined limit of the PR the book creates an alert.
4. The alert is published to the Alert Queue.

We introduce two new actors to manage the books and tag incoming events.

- The Event Router will notify the Book Manager about incoming PRs. It tags each event with the PR id it's related to and publishes the events to the Work Queue.
- The Book Manager keeps a roster of open books. Once it receives a notification from the Even Router it will check it's roster. If the PR id it received from the Event Router is new it will spawn a new book. The Book Manager starts a timer for each book spawned.
- Each Book consumes from the work queue. It only consumes events that are tagged with the PR id each book is keeping track of. Each Book keeps track of exactly one PR id and accumulates P events.
- If the accumulated sum of purchase events P exceed the limit defined in the PR the Book will publish an alert. After publishing an alert the Book dies immediately.
- Once the timer for a Book reaches the defined delta t the Book Manager sends it a kill command.
- A Book receiving a kill command finishes any pending processing and then dies.

The Auditor now has three internal actors:

1. Event Router
2. Book Manager
3. Book

Figure 3.2 illustrates the even flow between those components.
Figure 3.3 shows a system overview with component interactions.

3.2.1 Questions

1. How will the Book Router be able to keep up with the message rate?
2. Can we distribute the Book Router?
3. Can we implement the Book Router using a RabbitMq topic exchange?
4. How can we distribute and efficiently manage the creation of books?

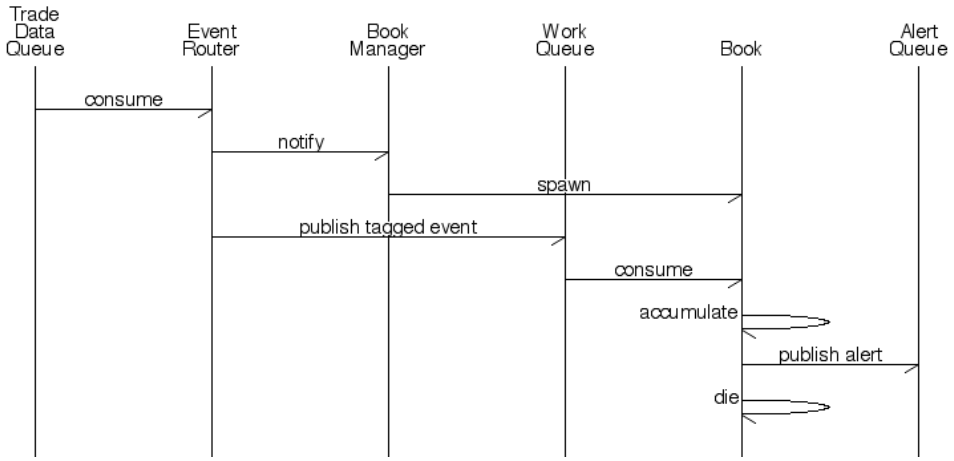

Figure 3.2: Auditor-1

5. Books will be very ephemeral. They need an ultra-light process abstraction?
6. Is routing purchase events P to open books efficiently the key for achieving optimal throughput?
7. How do we make sure a book does not generate more than one alert?
8. Is it easier to filter duplicates (if any) in the Channel Boy?

3.2.2 Assumptions

1. We do not need to restart books when a book process dies. Meaning if an open book dies it's state is lost.
2. For now throughput is more important than resilience.

3.3 Iteration 2

We now consider the most straightforward way to model the processing of events by using RabbitMq's routing exchanges.
The Event Router (there can be many) publishes each PR (purchase request) event to the Purchase Request Exchange. The Purchase Request Exchange routes each event to the Purchase Request Queue.
The Event Router (there can be many) implements the following functionality:

1. Consume from the Trade Data Queue and publishes PR events to the Purchase Request Exchange.
2. Consume from the Trade Data Queue and publish P events to the Purchase Exchange. The Book Router adds a routing header that contains the PR ID (initial order id). This header is then used by the Purchase Exchange (a topic exchange) to route the P event to the correct Purchase Queue.

The Book Manager (there can be many) implements the following functionality:

1. Consume the PRs from the Purchase Request Queue.
2. Check if a Purchase Queue with name `PQ-<PR ID>` already exists.
3. If a Purchase Queue with that name does not exist:
   - Declare a new Purchase Queue with a name like `PQ-<PR ID>`.
   - Create a binding with `<PR ID>` as routing key between the Purchase Exchange and the new Purchase Queue.
   - Spawn a Book.

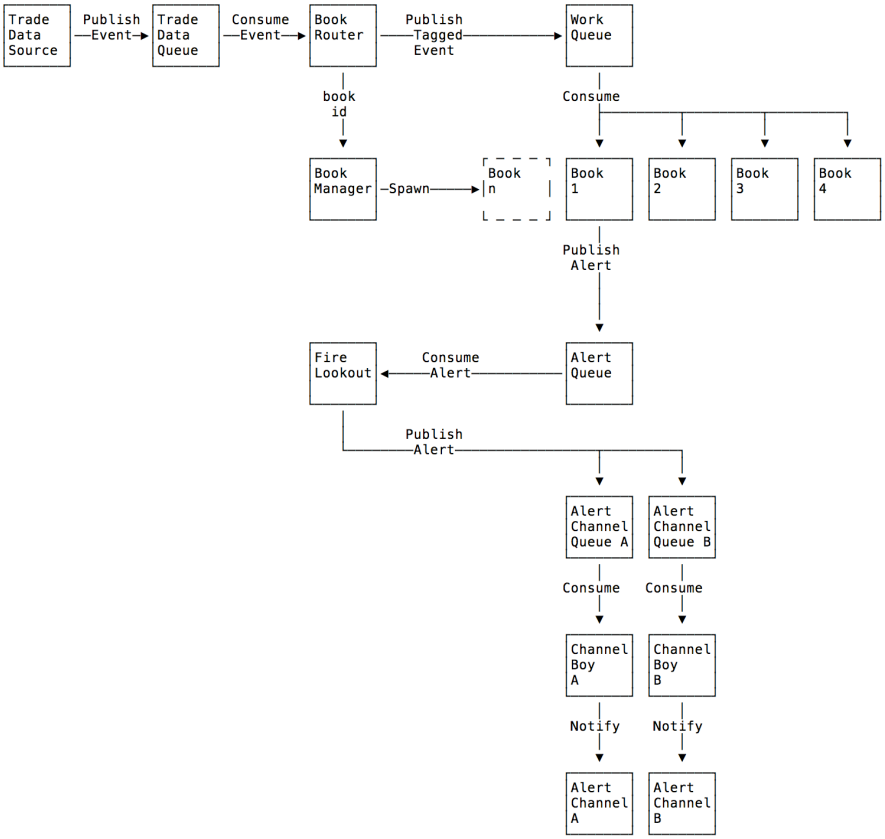

Figure 3.3: System Overview 1

The Book implements the following functionality:

1. Starts a timer at birth.
2. Subscribes to the Purchase Queue name `PQ-<PR ID>`.
3. Consumes P events and accumulates the number of purchased shares.
4. If the sum of all accumulated shares exceeds the limit defined for a PR the Book publishes an alert to the Alert Exchange.
5. Once the timer runs out it unsubscribes from the Purchase Queue and dies.

The functionality for the Event Router, Book Manager and Book live inside a component name Auditor. We can spawn multiple Auditors on multiple distributed nodes. All interactions between components are via queued messages. The Book Manager spawns Books but they auto terminate and no pruning or restart logic is necessary. Figure 3.4 illustrates the event flow of this design.
Figure 3.5 shows the updated system overview with component interactions.
u

3.3.1 Assumptions

1. Events are semi ordered sequences. A purchase request event always arrives before the related purchase events. This might not be true.

3.3.2 Questions

1. To reduce churn with queue creation / destruction maybe we should have one Purchase Queue per stock symbol? How would that affect management

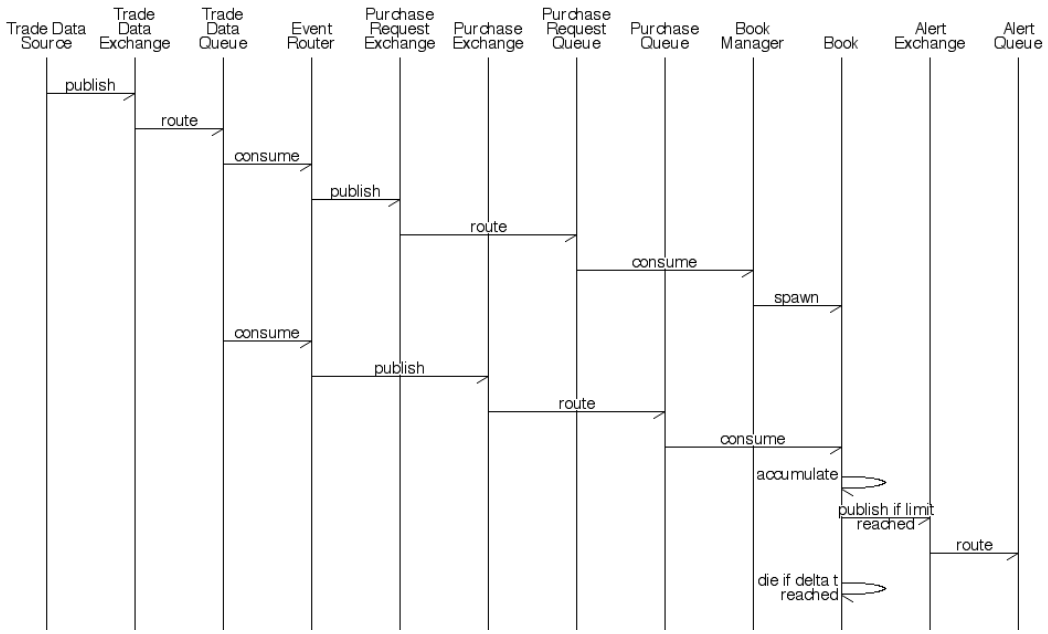

Figure 3.4: Auditor-2

of books?

2. None of the queues will survive a broker restart. Is that problematic?
3. There is a potential race condition when a P event arrives before we've created the Purchase Queue. How can we avoid that?
4. Can we automatically clean up the Purchase Queues if we set them to auto-delete? That would be removed them automatically once the Book unsubscribes.

3.4 Iteration 3

We need to prepare the development environment as described in section 5.1 and section 5.2.

3.4.1 Create the environment:

We start by creating the virtual Python environment:

```
mkdir $HOME/aldebaran
cd $HOME/aldebaran
mkvirtualenv aldebaran
workon aldebaran
```

Install the necessary libraries for the project:

```
pip install pika
pip install arrow
```

3.4.2 Create the exchanges and queues

```
# This file was auto-generated via org-babel-tangle in Emacs
```

```
# Do not modify this file manually. Instead modify the source
# in aldebaran.org and re-run org-babel-tangle
#
# Usage: ./declare.py -s localhost
#

import pika
import uuid
import arrow
import time
import sys
import argparse

def declare_exchanges(channel):
    channel.exchange_declare(exchange='Aldebaran.Trade_Data',
                             exchange_type='direct', durable=True)
    channel.exchange_declare(exchange='Aldebaran.Purchase_Request',
                             exchange_type='direct', durable=True)
    channel.exchange_declare(exchange='Aldebaran.Purchase',
                             exchange_type='topic', durable=True)
    channel.exchange_declare(exchange='Aldebaran.Alert',
                             exchange_type='direct', durable=True)

def declare_queues(channel):
    channel.queue_declare(queue='Aldebaran.Trade_Data', durable=True)
    channel.queue_declare(queue='Aldebaran.Purchase_Request', durable=True)
    channel.queue_declare(queue='Aldebaran.Purchase', durable=True)
    channel.queue_declare(queue='Aldebaran.Alert', durable=True)

def bind_queues(channel):
    channel.queue_bind('Aldebaran.Trade_Data', 'Aldebaran.Trade_Data' ,
      routing_key=None, nowait=False, arguments=None)
    channel.queue_bind('Aldebaran.Purchase_Request', 'Aldebaran.Purchase_Request' ,
      routing_key=None, nowait=False, arguments=None)
    channel.queue_bind('Aldebaran.Purchase', 'Aldebaran.Purchase' ,
      routing_key=None, nowait=False, arguments=None)
    channel.queue_bind('Aldebaran.Alert', 'Aldebaran.Alert' ,
      routing_key=None, nowait=False, arguments=None)

def main():
    parser = argparse.ArgumentParser(
             description='Declare exchanges and queues.')
    parser.add_argument('-s', metavar='rabbitmq', default='localhost',
                        help='The IP or DNS name of the RabbitMq server')
    args = parser.parse_args()
    myrabbit = args.s
    # connect to RabbitMq
    try:
        connection = pika.BlockingConnection(
                     pika.ConnectionParameters(host=myrabbit,heartbeat_interval=20))
```

```
    except Exception as err:
        print('Cant connect to RabbitMq. Reason: %s' % err)
        exit(1)
    channel = connection.channel()
    print 'Declaring exchanges and queues on the RabbitMq server.'
    declare_exchanges(channel)
    declare_queues(channel)
    bind_queues(channel)
    channel.close()
    connection.close()

if __name__ == "__main__":
    main()
```

3.4.3 Publisher

We start with a very simple publisher process. To simplify initial debugging we publishes with a frequency of 1Hz.

```
# This file was auto-generated via org-babel-tangle in Emacs
# Do not modify this file manually. Instead modify the source
# in aldebaran.org and re-run org-babel-tangle
#
# Usage: ./trade_data_source.py -s localhost
#

import pika
import uuid
import arrow
import time
import sys
import argparse
import json

def create_purchase_request():
  fields = {}
  fields['id'] = str(uuid.uuid4())
  fields['#shares'] = 100
  fields['client'] = 'Goldman'
  fields['stock'] = 'AAPL'
  return fields

def create_purchase(order_id):
  fields = {}
  fields['id'] = str(uuid.uuid4())
  fields['initial_order_id'] = order_id
  fields['#shares'] = 30
  fields['client'] = 'Goldman'
  fields['stock'] = 'AAPL'
  return fields
```

```
def publish_purchase_request(channel):
    pr = create_purchase_request()
    print ('Purchase Request: %s' % json.dumps(pr))
    channel.basic_publish(exchange='Aldebaran.Trade_Data',
                          routing_key='', body=json.dumps(pr))
    return pr['id']

def publish_purchase(channel, order_id):
    p  = create_purchase(order_id)
    print ('Purchase: %s' % json.dumps(p))
    channel.basic_publish(exchange='Aldebaran.Trade_Data',
                          routing_key='', body=json.dumps(p))

def publish(channel):
    order_id = publish_purchase_request(channel)
    publish_purchase(channel, order_id)

def main():
    parser = argparse.ArgumentParser(
             description='Loop sending events to RabbitMq with at a rate of 1Hz.')
    parser.add_argument('-s', metavar='rabbitmq', default='localhost',
                        help='The IP or DNS name of the RabbitMq server')
    args = parser.parse_args()
    myrabbit = args.s
    # connect to RabbitMq
    try:
        connection = pika.BlockingConnection(
                     pika.ConnectionParameters(host=myrabbit,heartbeat_interval=20))
    except Exception as err:
        print('Cant connect to RabbitMq. Reason: %s' % err)
        exit(1)
    channel = connection.channel()
    print 'Sending event to RabbitMq server. To exit press CTRL+C.'
    while ( 1 ):
        publish(channel)
        time.sleep( 1 )
    channel.close()
    connection.close()


if __name__ == "__main__":
    main()
```

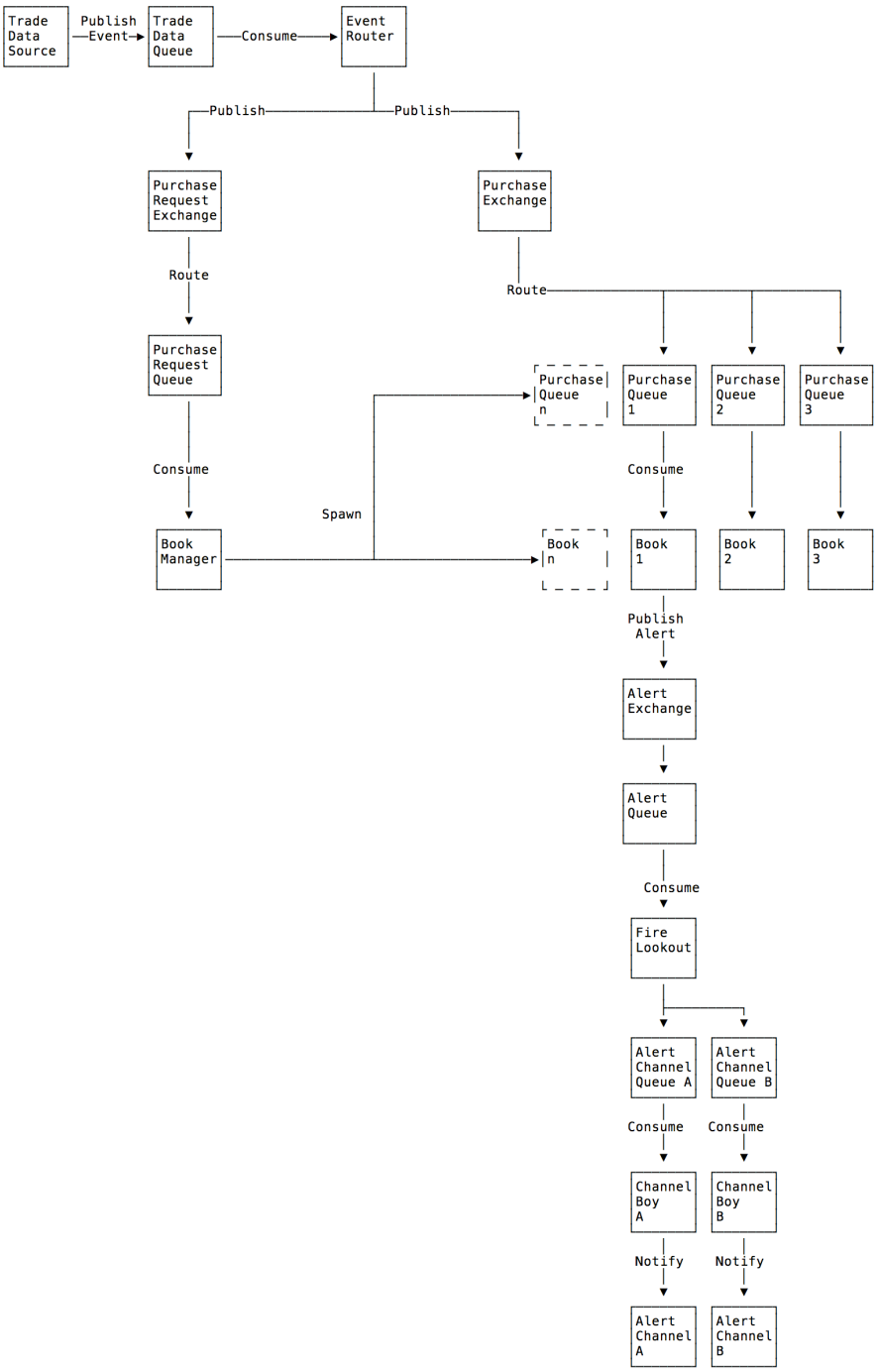

Figure 3.5: System Overview 2

4 Open Questions

- Scaling Up
- Resilience
- Minimizing Coordination
- Deploy / Release Tooling
- Live Debugging/ Patching
- Distributed Monitoring

5 Appendix

5.1 Setup RabbitMq

We describe the necessary steps to prepare an environment on OSX for RabbitMq. More information regarding the Firehose Tracer can be found here: Tracer
The Firehose Tracer enables the tracing of all messages that get published to exchanges and are consumed from queues.

5.1.1 Install RabbitMq

```
brew update
brew install rabbitmq
ln -sfv /usr/local/opt/rabbitmq/*.plist ~/Library/LaunchAgents
launchctl load ~/Library/LaunchAgents/homebrew.mxcl.rabbitmq.plist
```

5.1.2 Activate RabbitMq Admin Plugin

```
rabbitmq-plugins enable rabbitmq_management
```

5.1.3 Activate the Firehose Trace

Keep in mind that activating tracing is not presistent. If the server goes down or gets restarted we need to activate tracing again.

```
rabbitmqctl trace_on
rabbitmqctl list_exchanges
```

Activate plugin:

```
rabbitmq-plugins enable rabbitmq_tracing
rabbitmq-plugins list
```

5.2 Setup Python Environment

Install git-core:

```
brew install git
```

5.2.1 Install Virtualenv

Execute the following commands:

```
brew install python --framework
pip install --upgrade pip setuptools
pip install --upgrade virtualenv
mkdir ~/.virtualenvs
pip install --upgrade virtualenvwrapper
```

5.2.2 Install Realpath

The current `virtualenv` has a dependency on `realpath` that is not satisfied on OSX. Do the following to fix this:

```
brew tap iveney/mocha
brew install realpath
```

5.2.3 Configure

Add the following to `$HOME/.bash_profile` or `$HOME/.zprofile`:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python
export WORKON_HOME=~/.virtualenvs
source /usr/local/bin/virtualenvwrapper.sh
```

5.3 RabbitMq Tuning

RabbitMQ's queues are fastest when they're empty. When a queue is empty, and it has consumers ready to receive messages, then as soon as a message is received by the queue, it goes straight out to the consumer. In the case of a persistent message in a durable queue, yes, it will also go to disk, but that's done in an asynchronous manner and is buffered heavily. The main point is that very little book-keeping needs to be done, very few data structures are modified, and very little additional memory needs allocating. Performance of Queues
The main factor is consumer performance. Keep those queues empty!