



DEGREE PROJECT, IN MASTER'S PROGRAMME COMMUNICATION SYSTEMS
SECOND LEVEL

STOCKHOLM, SWEDEN 2015

Message-oriented Middleware for Scalable Data Analytics Architectures

NICOLAS NANNONI

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



Master's Thesis
KTH – Information and
Communication Technology School

Examiner
Markus Hidell (ICT)

Supervisor
Marko Helin (Accedo)

2015-02-13

Message-oriented middleware for scalable data analytics architectures

Author
Nicolas Nannoni
(Nicolas.Nannoni@accedo.tv | Nannoni@kth.se)

Abstract

The democratization of Internet allowed many more people to use online services and enjoy their benefits. The traffic towards websites became tremendous those recent years, especially with the apparition of social networks. Mobile application, televisions and other non-computer devices also get connected to the Internet and use it to provide services to the end-users: Video on-demand, music streaming and so on. These applications rely on powerful backend servers that handle the requests made by devices and provide statistics and metrics about application usage. These metrics can be generated by aggregating the access logs (e.g. HTTP requests log), logs that are potentially extremely large. Big data tools and analytics, providing a way to handle this huge number of records, come then in hand, as typical client-server architectures, with a single database storing all the data, reach their limits in terms of performance and capacity. Data duplication, combined to dedicated and specialized databases storing it, is the key to efficient data handling.

How to fill up these databases in an elegant, efficient and scalable manner is the remaining question, and message-oriented middleware may be a viable answer. This project aims at exploring the capabilities of such middleware, identifying what are the benefits and the drawbacks in using them and presenting how they can be integrated in a real-world application that needs to aggregate events and logs on a large scale. Apache Kafka and RabbitMQ, two message-oriented middleware, are benchmarked and compared, on both performance metrics and qualitative criteria. A fully working proof-of-concept (of an already-existing industry product modified to use a message-oriented middleware and a specialized data warehouse system) is developed and presented, to conclude on the usefulness of message-oriented middleware when designing scalable data analytics architectures.

Sammanfattning

Demokratiseringen av Internet har tillåtit många fler att använda online-tjänster och deras fördelar. Trafiken till webbsidor har blivit enorm de senaste åren. Speciellt i och med de sociala nätverken. Mobil-applikationer, TV-apparater och andra enheter ansluter sig i allt större omfattning till Internet och tillhandahåller tjänster till slutanvändare: Video On-Demand, strömmande musik o.s.v. Applikationerna förlitar sig på kraftfull infrastruktur som kan hantera de förfrågningar enheterna gör och tillhandahålla statistik och mätetal om applikationernas användning. Dessa mätetal kan skapas genom att aggregera access-loggar (ex. HTTP-loggar). Dessa loggar är potentiellt väldigt stora. Så kallade Big Data-verktyg kan lösa problemet med att hantera denna stora mängd data. Typiskt är dessa verktyg klient-server-arkitekturer med en enskild, central databas som lagrar all data. Dessa databaser har i regel begränsningar när det gäller prestanda och kapacitet.

Duplicering av data kombinerat med en dedikerad och specialiserad databas är nyckeln till en effektiv lösning på detta problem. Frågan är hur man på ett effektivt, elegant och skalbart sätt fyller dessa databaser med information. Här kan meddelande-baserad mellanprogramvara vara en lösning. Det här examensarbetet syftar till att granska hur sådan mellanprogramvara kan integreras i en applikation som används i branschen idag och som behöver aggregera stora mängder loggar. Apache Kafka och RabbitMQ, som är två meddelande-baserade mellanprogramvaror, granskas och jämförs. Prestanda och effektivitet av lösningarna testas. En fullständig prototyp skapas. Den baseras på ett befintligt system och ändras för att använda meddelande-baserad mellanprogramvara och ett specialiserat Data Warehouse-system. Slutligen dras slutsatser om meddelande-baserad mellanprogramvara är effektivt när man vill skapa ett skalbart system för aggregering av loggar.

Contents

1	Introduction	7
1.1	Background	7
1.1.1	Accedo Appgrid	7
1.2	Motivations	8
1.3	Problem statement	9
1.4	Content of this document and methodology	9
1.5	Outcomes, ethics and sustainability	10
2	Message-oriented middleware: introduction	12
2.1	What is a message-oriented middleware?	12
2.2	Typical use cases for message-oriented architectures	13
2.3	General benefits of message-oriented architectures	14
2.4	General disadvantages or difficulties in implementing message-oriented middleware	16
3	Case study: Accedo Appgrid	19
3.1	Current Appgrid architecture	19
3.2	API overview	20
3.3	Log, event generation and management	20
3.4	Original log and event insertion process	21
3.5	Introducing a message-oriented middleware in Appgrid architecture	22
3.6	Message-oriented log and event insert process	23
3.7	Possible improvements to the basic message-oriented log insertion process	24
4	Choice of a message broker	26
4.1	State of the art in message-oriented middleware	26
4.2	Apache Kafka	28
4.2.1	Overall structure	29
4.2.2	Main features	30
4.2.3	Set-up and management	33
4.3	RabbitMQ	33
4.3.1	Overall structure	34
4.3.2	Main features	36
4.3.3	Set-up and management	38
4.4	General feature comparison between Apache Kafka and RabbitMQ	39
4.5	Preliminary conclusions	41
5	Implementation in Appgrid and test	42
5.1	Motivations and goals	42
5.2	Code architecture	42
5.2.1	Core Appgrid modifications	42
5.2.2	Consumer process	44
5.3	Message structure and protocol	47
5.4	Queue/topic architecture	50
5.5	Overall test architecture and load generation tool	51
5.6	Test scenario	53
5.7	Load profile	54
5.8	Tested configurations	55
6	Test results	56
6.1	Testing hardware and conditions	56
6.1.1	Java Virtual Machine and JIT compilation	56
6.1.2	Test reliability	57
6.1.3	Producer, broker and consumer settings	57

6.2	Followed metrics and instrumentation	57
6.3	Data analytics methods	59
6.4	Appgrid performance results	61
6.5	Broker specific results.....	62
6.6	Broker and consumer performance.....	65
6.7	Test conclusions.....	68
7	Leveraging the advantages of Message-Oriented Middleware for data analytics	69
7.1	Current aggregation process in Appgrid	69
7.1.1	Problems with this solution	70
7.2	Proposed solution: split the work	71
7.2.1	Amazon Redshift.....	72
7.2.2	Database architecture	73
7.2.3	Aggregation processor.....	74
7.3	Performance testing architecture.....	75
7.4	Test configurations and checking process.....	77
7.5	Test results and conclusions	77
7.5.1	Single vs. multithreaded aggregator process	78
7.5.2	Disk space usage.....	79
7.5.3	RabbitMQ and Amazon Redshift resiliency.....	80
7.5.4	General conclusion.....	81
8	Conclusion and future work.....	82
8.1	Advantages and perspectives offered by message-oriented middleware	82
8.2	RabbitMQ vs. Kafka	83
8.3	Event-oriented and service-oriented architecture	84
8.4	Future work.....	86
9	Glossary	88
10	List of figures	92
11	List of tables	93
12	References	94
Appendix A	— Apache Kafka – Quick reference	101
1.	Install Kafka on the broker.....	101
2.	Edit the brokers' and Zookeeper preferences	101
3.	Start Zookeeper and Kafka	101
4.	Updated ZooKeeper CLI tool	102
5.	Management operations.....	102
6.	A Web UI for Kafka: kafka-web-console	103
Appendix B	— RabbitMQ – Quick reference	104
1.	Install RabbitMQ on the broker	104
2.	Edit RabbitMQ preferences.....	104
3.	Start and stop RabbitMQ.....	104
4.	Enable and use the management UI.....	105
5.	Python script to parse queue logs	105
Appendix C	— Tsung – Quick reference	107
1.	Download and install Tsung.....	107
2.	Launch a test.....	107
3.	Generate test reports.....	107
4.	Frequently asked questions	108
5.	Example Tsung XML test scenario description	112
6.	Helper scripts (Bash) to install Tsung, and deploy and run tests.....	113

Acknowledgements

Now that this project is done, I can take some steps back and realize where I am now, and who I am accountable for this. I would like first to thank Fredrik Sandberg and Marko Helin from Accedo for their genuine trust from the beginning, that has been renewed recently again, as I became part of their great, permanent work team. Alexej Kubarev and Erik Gulliksson were also of great help and support throughout the thesis and I am therefore very grateful to them! I also wish to thank Markus Hidell from KTH, for the excellent courses he gave me at the university, and also for having accepted to be my examiner for this thesis.

Moreover, I would like to thank my parents, my sister and my brothers, back in France, for supporting me even though I am not always close to them, and available as the first-born child should be. Finally, I want to specially thank my dear friend Thibault Doublier, not only for bearing with me at home, but also for providing me his complete support and highly valuable inputs in the projects in which I am involved.

Stockholm, November 25, 2014.

1 Introduction

1.1 Background

Big data is an expression that is among the most popular nowadays in the computing world. While not being a new concept, as some financial companies like MasterCard relied for decades on large data warehouse, it has gained in the recent years a large interest from the industry [1, Sec. 1]. The quantity of data one can easily and cheaply gather, store and therefore be willing to analyse became indeed enormous. Traditional tools are not powerful enough to analyse such large data sets, but “big data isn’t just a description of raw volume”: identifying and producing meaningful data out of it is the main challenge. Information one can extract from carefully crafted queries, run against enormous sets of data, can help decision-makers, advertisers, researchers or e-merchants to improve their knowledge of the field they are working in.

Accedo, the company at which this Master’s thesis was carried out, is one of the leaders application and software providers in the IPTV and SmartTVs area. Accedo is developing different solutions, used by customers such as media companies, telecom operators and device manufacturers, to help them seamlessly deliver media contents on different platforms (smartphones, tablets, gaming consoles, etc.). One of their solutions, Accedo Appgrid, is the main focus of this thesis, and is presented hereafter.

1.1.1 Accedo Appgrid

Appgrid is a product that acts as a sort of Content Management System (CMS) for mobile and SmartTV applications. It can store in the cloud metadata (i.e. settings stored using a key/value scheme) and assets (e.g. images, short movies) that different applications on different platforms can fetch through a simple REST HTTP API. Appgrid therefore allows content providers to remotely update what they serve to their clients at one point, for all the different architectures, platforms and operating systems that clients might run the app on. The data available in the native applications they develop can be changed without requiring any change and republishing (avoiding revalidation process as well, for Apple for instance) of each and every application, thus greatly lowering the time between a content change and its delivery to the end users.

Figure 1 (on the next page) presents the overall landscape of an application using Appgrid: on the right-hand side, all the devices that may be used to run the application (no matter the platform: Android, iOS, SmartTV, etc.) contact the Appgrid API server in the cloud via HTTP requests to fetch assets and metadata. Each device identifies itself by requiring a session key (that it obtains by sending both the application key, identifying the application and its specific platform, and a unique device ID). Application developers implement the logic required to request a session key and query for assets in the core code of their apps.

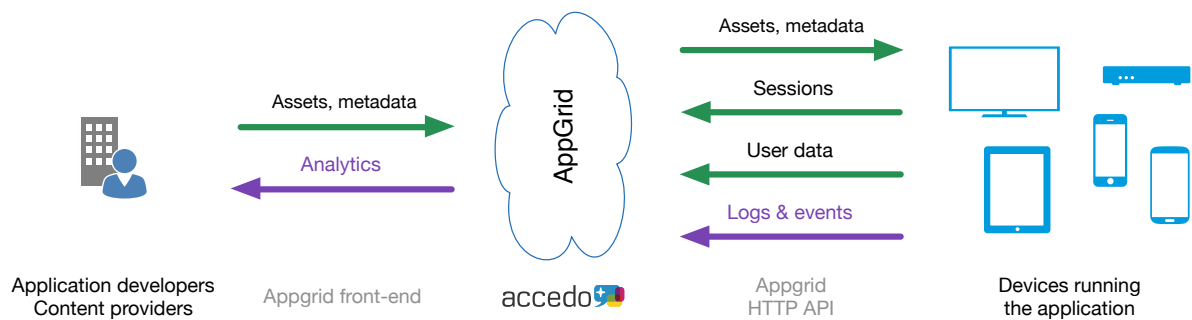


Figure 1 - Accedo Appgrid solution overview

Besides this CMS-like function, Appgrid also acts as a powerful tool for statistics and analytics. Indeed, Appgrid features an administration interface (left-hand side of Figure 1) that can help application and content providers to analyse application usage with a huge variety of criteria, including geographical and platform distribution. Events are triggered by the API itself (e.g. when an asset like “image.jpg” is requested) and are then aggregated to provide content editors meaningful indicators about their applications (like “how many people use my application?” or “which country uses it most?”). Application developers may also trigger events themselves by sending “logs” to the API through a specific HTTP request. Those logs may include any kind of information (e.g. an error encountered by the app, a notification telling that the user disabled one specific function, etc.), flow through the same API, and are stored in one, single database instance, in the current architecture.

1.2 Motivations

Finding tools, strategies and architectures that are powerful, resilient and efficient enough to handle, store and then analyse large stream of data is what is really valuable for a company or an individual. And one of the keywords that should be kept in mind in this quest is “multiplicity” [1, Sec. 2]. Indeed, a single tool, a single database is not likely to provide the best answer to all the queries one might want to perform. Therefore, single-centred architectures, with a single “source of trust”, where all the data is stored in a single database, is probably not the way to go, and some duplication of data must be performed, dispatched in various, specialized tools.

This is where message-oriented middleware, the essential part of the thesis subject, comes onto the scene [2]: cleverly inserted into a server architecture, this “application-level” postman system may help to distribute the data to the right place, the right storage system efficiently. And eventually contribute to design sustainable big data architectures.

Writing the queries that would analyse the data sets is a task left over to data scientists. Quite often, from a theoretical point of view, the question to be answered by a query on large sets is very simple. In the thesis, they are like “how many people downloaded the logo of this company in September?” or “in which country is this feature mostly used?”. There can be workflows and situations in which queries get complicated, but that should not affect the overall data architecture needed to process them.

1.3 Problem statement

The current Appgrid architecture is already capable of performing queries on the large amount of data it is handling. However, the number of end-users using the service is expected to soar in the next coming months, thus making the gathered data volumes to swell a lot. A more sustainable and scalable solution must be found to ensure that Appgrid offers top-grade performance, no matter the number of customers. This is the problem addressed by this thesis.

The hypothesis is that message-oriented middleware can allow a clear task separation between the application main business logic and the log/event management process. This would enable the use of new and flexible tools to better aggregate the events, while offloading this operation from the main application. The thesis explores the implementation of message-oriented middleware in Appgrid, tests it in real conditions and concludes on this question.

1.4 Content of this document and methodology

This report presents quickly what is Appgrid, introduces in section 2 the notion of message-oriented middleware, and explains why it is interesting, theoretically speaking. It then focuses in section 3 on the Appgrid case and study how the original architecture may be updated to take advantage of such middleware.

The report goes onwards and presents in section 4 two data busses (RabbitMQ and Apache Kafka), and then section 5 exposes the test architecture and the modifications made in Appgrid to test them in real conditions. The comparative and quantitative tests carried out in section 6 helped to draw preliminary conclusions on those systems. Perspectives opened by the tests performed and which ones could be done in a future study are also presented. Finally, section 7 provides a concrete and complete example of a sustainable big data architecture relying on the concepts presented in the report, with quantitative comparison of their benefits.

Comparing two message-oriented middleware through quantitative tests was justified by the need to get real figures, and assess the stability of both tools under production-like conditions. As presented in section 4.1, some benchmarks of these systems already exist, but these are brute tests focusing on the broker itself. They neglect the impact of the libraries used by the processes around the broker that interact with it. Based on trustworthy figures and a thorough analysis of the qualitative criteria (e.g. quantity and quality of documentation, ease of use, etc.), choosing the best broker could be done and the outcome used to achieve the ultimate goal of the thesis in section 7 : building a complete example of an optimized data aggregation process.

For the sake of reproducibility, the reader may also find appendixes at the end of the document, which should give him enough material to test the presented solutions and provide him support with the problems the author went through when trying them. Also, a glossary is provided page 88 and can be used as a quick reference for specific words used in this document.

1.5 Outcomes, ethics and sustainability

This report presents message-oriented middleware from a general point of view and then uses Appgrid for a case study, to test those middleware in real conditions. A complete proof of concept has been developed, and a fully working Appgrid version, relying on the presented systems, has been delivered. While the outcomes of the tests in terms of quantitative results may not be applicable for other applications, the overall conclusions should help any individual to choose one of the message busses presented if the workflow is somehow similar to the one presented. Section 7 also presents a high-level overview of all the advantages that the concepts presented in the report can give to system architects and also non-technical decision makers, with a simple cost analysis and system metrics comparison.

The thesis focuses on the technical questions linked to architecture design more than pure calculation and query contents. As such, it does not go any deeper in how do databases process queries, what they store, and what could be inferred from what is stored, using machine-learning techniques and other prediction algorithms. These ethical issues are more related to the use one can make of the systems that are presented here than the systems themselves. They are however critical and should not be dismissed: it has been shown that big data analytics can for instance be used to “create instantaneous and surprisingly detailed psycho-demographic user profiles” using simple data extracted from social networks [3], [4]. Storing data relative to medical records or tracking user behaviour on the Internet also allows one to draw conclusions on people’s personalities, threatening their privacy [5], [6].

Appgrid is tracking users’ actions in applications via the calls they are making toward the API. Each user is uniquely but non-personally identified. The data is aggregated in such a way that links between individual users cannot be inferred from the final results (in a similar process to what is achieved with *k-anonymity* [7]). Proper information should be given to the end-users, telling them that their actions are being tracked and then processed by an automatic system. The metrics provided by Appgrid regarding application usage are among the most valuable features of the platform for content providers. Therefore, having a comprehensive set of tools and data aggregation processes, whose outputs are general but meaningful enough while avoiding individual user tracking is a real challenge. Securing the whole architecture, to also prevent unwanted data leaks or intrusions, is needed when managing such a gigantic amount of data. These matters are not discussed in the thesis but need to be taken into account when releasing such analytics systems.

Another point that is essential to keep in mind in the global climate-change situation the world is facing now is that storing and crunching this gigantic amount of data is highly energy consuming. In the United States, 2% of the total electricity used is spent to power data centres [8]. In Sweden, the government has been recently proposing to lower taxes on electricity bills to attract more IT companies willing to build such server farms in the country [9]. Even though one data centre owned by Facebook and implanted in Luleå consumes as much electricity as 16,000 households [10].

The environmental footprint of such data centres is therefore far from being neutral, and efficiently designing the applications that run on them may affect to some extent the energy they consume. Making the crunching processes more efficient would be not only better for the environment, but would also decrease the costs of operation of such data centres, which are tremendous [11]. Carefully thinking the whole process of data collection, storage and crunching, so that they take less time to do the same job, is therefore not only good for the end-users' satisfaction, but also for the benefits they give in terms of energy efficiency and cost reductions.

2 Message-oriented middleware: introduction

The purpose of this section is to give a wide introduction to message-oriented architectures. No prior knowledge in the field should be required to understand this part. If the reader feels comfortable with these matters, he can proceed directly to section 3 page 19 that deals with the implementation of those systems in a real application.

2.1 What is a message-oriented middleware?

The whole point of the thesis is to demonstrate the use of a message-oriented middleware, and how it may allow typical backend architectures to scale more easily, while being even more extensible. But what is it?

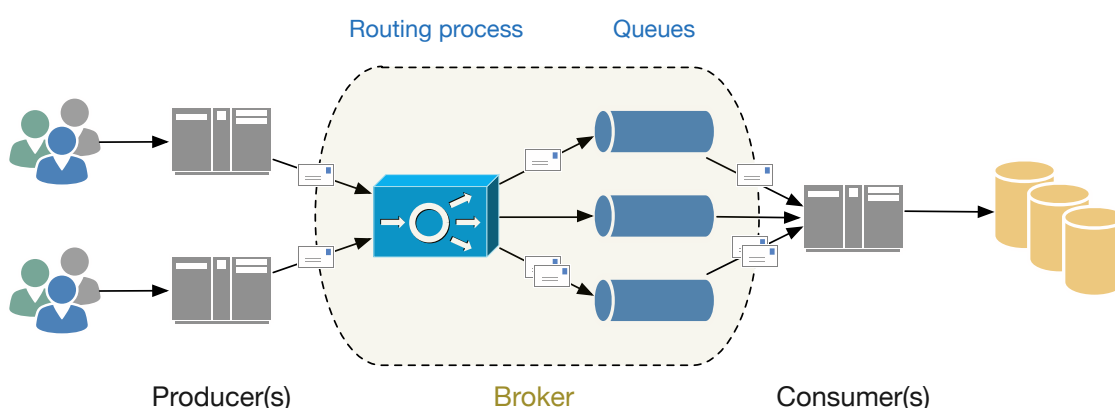


Figure 2 - Overview of a message-oriented architecture

As the name implies, a message-oriented middleware (or data bus) is a piece of software that will deal with messages, those messages being any kind of data (usually, a raw string of bytes), wrapped into a structure that features a very simple header. Messages are sent by one or many processes called “producers” toward the message-oriented middleware that is usually called “broker”. The broker will not process the messages (i.e. it will never read their content) but may route them into zero, one or more queues according to the information enclosed in the header of the messages and its own routing table. The queues are stored locally in the broker: the routing table term does not refer to any kind of network stuff here.

Those messages were sent by the producers to get processed (e.g. a calculation must be done, a file should be stored, a record should be inserted in a database). The processes responsible for opening the messages and performing real actions with their content are called “consumers”. They will (depending on the kind of data bus used) either get messages pushed by the broker to them, or fetch some of them by regularly querying the broker. In the most simple use case, messages are then discarded once processed, and consumers carry on with the next ones.

Please note that the messages this document talks about have nothing to do with typical emails. Messages here are sent by a server process running on a machine to another machine. They are likely not to be human-readable at all. A complete explanation for readers uncomfortable with this notion can be found in section 5.3 page 47.

2.2 Typical use cases for message-oriented architectures

One clear distinction that should be made between typical direct process-to-process interaction and interaction involving a message broker is that all the operations are inherently asynchronous: the producer pushes a message to the broker but has no guarantee at all regarding the time between it posts the message and its consumption. Moreover, it will (in typical cases) not be notified when the operation has eventually been done.

Therefore, those messaging and queuing routines presented in the previous section cannot be used for operations that require (immediate or not) responses. Reading a value is in most cases not a deferrable operation. But writing a value may be. Indeed: if the value to be inserted is not needed (at the time its creation is requested and in the near future) by any other process, its insertion can be delayed and queued for a while without any trouble.

The most obvious use of such a deferred write operation is when written entities are logs [12, Sec. 12.5]. A log entry is a standalone entity: no other entities have a key pointing to a log entry. Besides, it is not vital (in most production environments) for an application to publish its logs in real-time. The *only* requirement is that the data should be eventually written down somewhere. That is precisely what a message-oriented architecture can guarantee: that provided a consumer is available, a message will get processed, but without any indication on the time it will take.

Other use cases may include email sending (note however that messages are not more equivalent to emails than in the previous example: messages would contain *email jobs*, not emails) or parallelized and distributed computations [12, Sec. 12.1, 12.3]. Bi-directional communication between two asynchronous processes may also fit these architectures, as depicted on Figure 3.

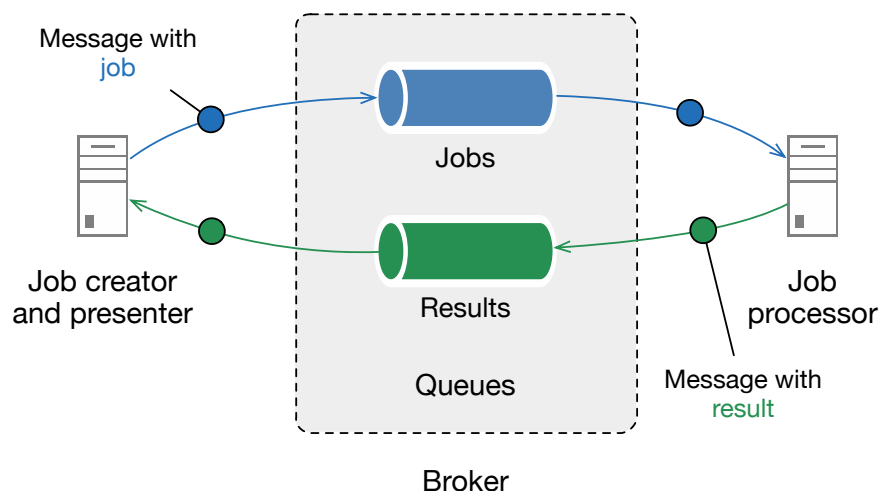


Figure 3 - Full-duplex communication between processes

Here, jobs and results are stored in queues in the same broker, which is not an absolute requirement. However, it should be kept in mind that all the operations are done asynchronously, and that there is therefore no guarantee that a successfully queued job will be even processed. The job creator must also have a way to match the results it receives with the jobs it sent to present them in a correct way when the answer eventually comes. Some message brokers give a way to tag messages and would thus allow such identification, but that is not part of the default feature set of brokers. This specific use case (that may account for event-driven architecture – EDA – or even staged event-driven architecture – SEDA [13], [14]) may be explored in a future study, building upon the findings of this work.

2.3 General benefits of message-oriented architectures

Besides the fact that a message-oriented architecture enforces a true and reliable asynchronous mode of operation of the systems it is applied to, it offers various additional advantages:

- **Flexibility and extensibility**

The broker can handle more than one type of messages, can have one or more queues and can duplicate messages to multiple queues if needed. But the most powerful functionality it gives to system architects is its total independence regarding the consumer and producer processes: these processes can be typically written in a wide variety of programming languages. Provided that they all use the same protocol, the same structure to define the content of the messages, they can seamlessly communicate with each other.

Producers can be of various kinds, and consumers as well. Multiple consumers can eat up the data from one queue on a Round Robin-like basis (i.e. only one consumer gets a copy of each message) or they can also do different tasks with the same input (i.e. all of them receive all the messages). This is a very powerful feature, as it can help replicating data in real-time to different data storage systems, for instance (see Figure 38 page 83 for an example).

- **Scalability**

Any number of producers may fill any number of queues, from which any number of consumers can simultaneously or not consume the data. It makes horizontal scaling very easy: the more HTTP requests arrive, the more public facing backend servers can be set up, sending data to the broker. The same can be done with the consumers, to increase the consuming rate of messages. The broker can also be clustered, to perform both load balancing and replication. However, the instances with which the producers and consumers would have to exchange data (e.g. databases, file storage systems) may get overloaded too. As such, they should be able to scale as well, to really give an architecture implementing a messaging middleware the power to scale up entirely.

- **Resilience and fault tolerance**

Such architectures can be stronger against software and hardware failures. Once the message is pushed to the broker, it can take care of replicating it and storing it on the disk to prevent any bug that may cause the broker to crash before it could deliver the message. The same applies to consumers: if they encounter a problem, they may close the connection they maintain with the broker, and just stop to process messages until they are restarted, or until another one takes up the task. Consumption acknowledgements can be used on the consumer side, to let the broker know when a message has been processed, for it to know when to redeliver it in case of failure of a consumer.

- **Burst management and batching**

Traffic bursts occur quite often on the Internet, and may overload for a short while some servers. By relieving them of some operations that can be deferred, the overall amount of resources dedicated to handle connections can be increased without changing the whole server code. Message-oriented middleware are designed to have very high throughput capacity, so that they could have a “shock-absorber” role.

This “shock-absorbing” role also gives freedom to system architects on the consumer side: namely, different consumers can be set up, consuming the same set of messages (all receive all messages), but possibly not at the same pace, and even not at the same time. It provides a great way to transform costly one-shot operations (like database insertions) into batch operations, by waiting enough messages to come before pushing them to a database.

Also, if a process is slower than another (e.g. one process adds the data into a MongoDB instance, while the other does it in an Hadoop cluster), they will not interfere or hinder each other and will perform the best they can.

- **Process isolation**

Message-oriented architectures enforce a de facto task separation. Processes that are not time dependent, critical and/or that must be replicated are better handled outside of the main application server. Dedicated small processes take care of the tasks independently, while the broker acts as a queuing and scheduling system.

- **Streamlined maintenance, management and delivery process**

Together with the process isolation and the resilience it gives, a message-oriented architecture is also easier to manage: tasks can be on the fly redirected toward new consumers, which can be turned off and on without impacting the application server in production. New consumers can be installed while the old ones are torn down without any downtime: the broker in between acts as a queue, continuing to receive and keeping the messages from the producers, while it waits for new consuming processes to be brought up.

2.4 General disadvantages or difficulties in implementing message-oriented middleware

The basic idea behind architectures relying on server messaging is not difficult to grasp, but can be somehow hard to implement in a real world application. Some other problems might arise too:

- **Multiplication of instances and applications to manage**

As a direct implication of the process isolation these architectures give, the total number of machines, application code and instances to watch increase. While a large application, with medium to heavy data load and possibly data replication need, would probably enjoy better the benefits of this more complex architecture, smaller applications might not find it useful enough to afford adding the required new instances to their workload.

- **Duplicated messages**

All broker systems on the market can lead to duplicated messages. That is, messages that get processed by a consumer that does not report to the broker that it has consumed the message, and therefore causes the message to get processed again (see Figure 4). Depending on the task accomplished by the consumers, it may create serious data integrity problems (e.g. duplicated entries in databases).

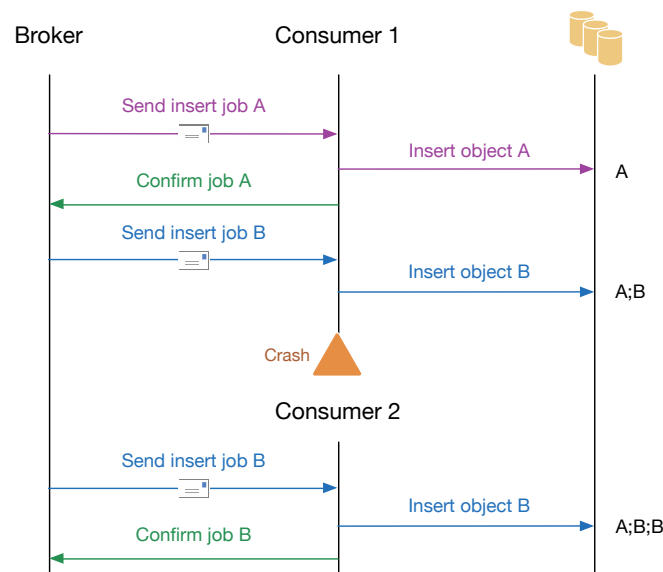


Figure 4 - Example of a scenario with duplicated messages

Note that the same problem can happen between the broker and the producers: depending on the system used (whether push-to-queue acknowledgments¹ are used/available or not), messages can get lost due to network problems, or a queuing acknowledgement can get lost and therefore cause a message to be pushed twice (or more) in queue.

¹ Also known as “publisher confirm”. See glossary page 88.

- **Ordering**

Depending on the system used, different guarantees (if any) are provided regarding the order in which messages are stored at the broker side, and consumed by the consumers. This must be taken into account when designing the protocol used to exchange messages if ordering is important. Including sequence numbers may help mitigating the problem for instance (but the question of its synchronization among all the instances may arise). Otherwise, deferring idempotent operations only (that is, operations that can be performed in any others and would still give the same results [15]) solves the problem by not creating it.

- **Code/process duplication**

Depending on the deferred process and what it has to achieve, the isolation of the different processes that messaging architectures provide may lead to code duplication. For instance, deferring an insert in a database that has to perform a query to populate a foreign key attribute prior to pushing data in a table may create new bottlenecks on the database, due to the number of simultaneous reads performed on the database by the different consumers. Moreover, if the data is not “ready to eat” in the queues in the broker, it is likely that all the different consumers that consume the same set of data (all receive all data to produce different results) will perform the same queries. Thus leading to resource waste and possible code inconsistency across the consumers.

- **Old and new data mixing**

The flexibility in terms of delivery and maintenance that message-oriented architectures give may also create problems. Assume that messages enclose an entity ready to be inserted in a database, and that a queue is full of those messages, without any consumer. Assume that the database model is changed (e.g. removing one of the field) and restart the same consumers. Then the insert operations are likely to fail or to create integrity problem, as the data is inserted “as is” by the consumers. Producers and consumers code need to be updated as well, and extreme care must be taken when dealing with inconsistent data queues. A clever yet simple message protocol design might help a lot to mitigate these issues (see section 5.3 page 47).

- **Induced delays**

The main purpose of implementing a message-oriented middleware is to completely separate asynchronous operations from the main application process. That is likely to induce a higher delay before completion, as the number of steps from the asynchronous operation triggering in the main application to its complete execution is higher. Moreover, there is no guarantee at all that a consumer is connected and ready to process data. This relative unpredictability may be a problem with near real-time operations. Having enough consumers and ensuring that they do not get overwhelmed can mitigate those delay issues.

- **“Moving” performance bottlenecks**

When designing asynchronous architectures, especially with messaging systems, one attractive goal is for sure the performance gain on the main application. Of course, deferring more processes to smaller and dedicated applications will relieve the main system, but this “magic” cannot replace a thoroughly thought architecture. Avoiding a bottleneck due to limited database insertion capabilities in the main application will not make it completely disappear: it will most likely show up again, in same or even worse proportions, in the consumer process that will do the same operation. However, consuming processes can be optimized to take advantage of the large amount of elements to process it may receive at the same time (e.g. use batch inserts in a database).

3 Case study: Accedo Appgrid

To really test and appreciate the performance and the features of specific message-oriented middleware, implementing it in a real application can be a good idea. This section presents the architecture that the target Appgrid solution currently relies on, and proposes a few modifications (with the help of message-oriented middleware) that can be implemented to improve the overall performance of the application, while testing the data busses systems.

3.1 Current Appgrid architecture

As presented in section 1.1.1, Appgrid is a cloud solution that enables content providers to serve assets and metadata to mobile devices and SmartTVs. These clients are at the top of Figure 5, which presents the overall Appgrid backend architecture. Devices send HTTP requests toward a single URL that is linked to an Elastic Load Balancer [16], which is an Amazon's implementation of an HTTP sprayer. It evenly distributes the requests load across two identical Appgrid API servers.

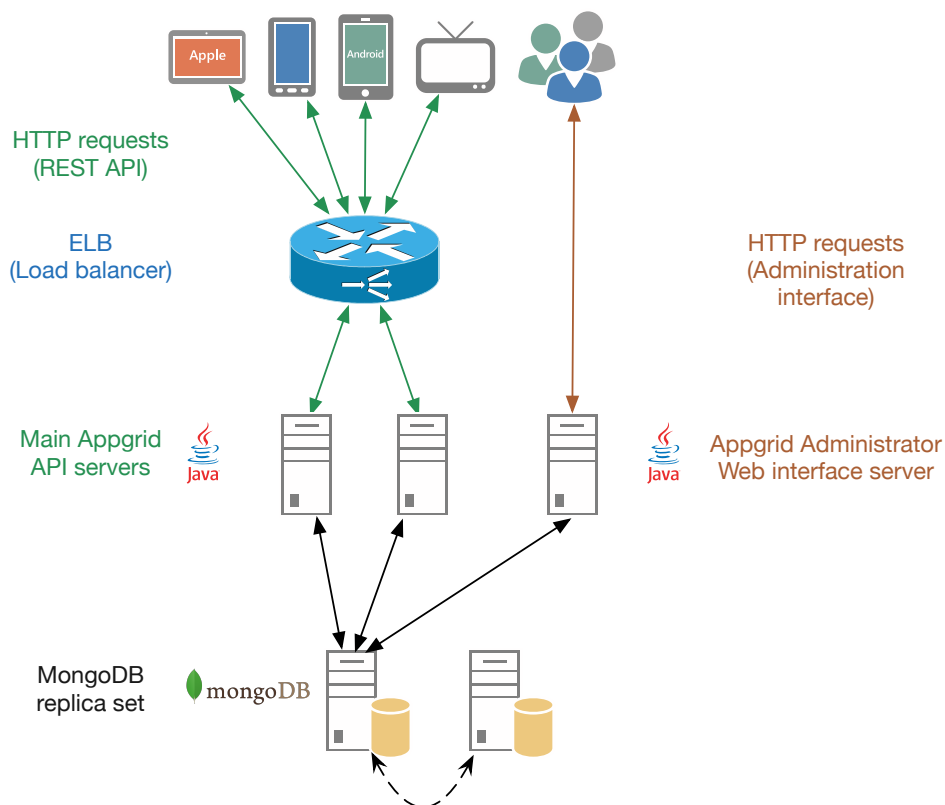


Figure 5 - Current Appgrid architecture

Appgrid is not limited to a pure API/backend system and also provides a frontend interface to allow content producers to change the assets they provide through the Appgrid API, and access statistics. This frontend is handled by a dedicated Appgrid server, which runs the exact same code as the two API servers, but only deals with frontend requests. Both the API servers and the administration frontend server share the same MongoDB database¹, and can therefore be in-sync by storing their state in it (e.g. sharing of session keys across the two API servers). The MongoDB database is set up in a master-replica scheme that allows seamless failover between the two MongoDB servers in case the primary fails.

3.2 API overview

Appgrid offers a convenient RESTful HTTP API to allow application developers to fetch metadata (key/value objects) and assets (e.g. films, images, music) in the cloud. Content providers define Applications in Appgrid, which are linked to an *applicationId* key that is used to uniquely identify each application when requests are made toward the API. An application that wants to interact with the API first requests a session key (that is, a unique random string that identifies a specific user session) and identifies itself by passing the *applicationId* and a unique identifier from the device that sends the request (to track returning users).

Using this session key, the application can make any kind of requests, asking for data or settings' values. These interactions are synchronous: the client makes the request and expects the response to come as quickly as possible, because it needs it to complete its task. It is a standard client/server model and it cannot be changed.

However, the API also features some so-called “fire-and-forget” methods. It means requests that clients can send, but that will not return any response other than a 200 HTTP code. These methods are usually triggering actions on the server, in which the client is not interested by the result. It just wants them to be done. Log and event pushing methods are among those “fire-and-forget” methods.

3.3 Log, event generation and management

Appgrid is not only a content management and delivery system, but also an analytics tool. It can give content providers information about application usage. To provide meaningful data to content administrators, a lot of data about API usage needs to be collected and aggregated. But prior to all this statistics and data analytics work, the API usage data must be generated and stored.

Appgrid has in its current architecture an internal event mechanism, which is triggered when some events arise (e.g. when a specific URL is hit, when a session key is created, etc.). They are ultimately pushed, together with some context information (e.g. the session and application key of the client that originated the request), to a specific collection in a log database in the MongoDB replica set.

¹ MongoDB is a NoSQL database. More information can be found in the glossary page 88.

While many of these events are triggered automatically in the Appgrid core code, some events can also be pushed manually by the applications running on the clients' devices to the API. Dedicated HTTP requests can for instance be used to store application events (e.g. when the application starts or closes). A single POST request, with the session key and the event type, may create an event entity that would get saved in a specific collection in the MongoDB replica set.

Finally, application developers may also use Appgrid as a log collector and push technical or usage information to a specific collection that they can then access via the administration frontend. Statistics on these logs can also be provided. POST requests are also used to push these logs to the database.

3.4 Original log and event insertion process

The application general process that handles events and log is described on Figure 6. Some simplifications have been made (the read operations in the database are not shown), but it gives the overall working process.

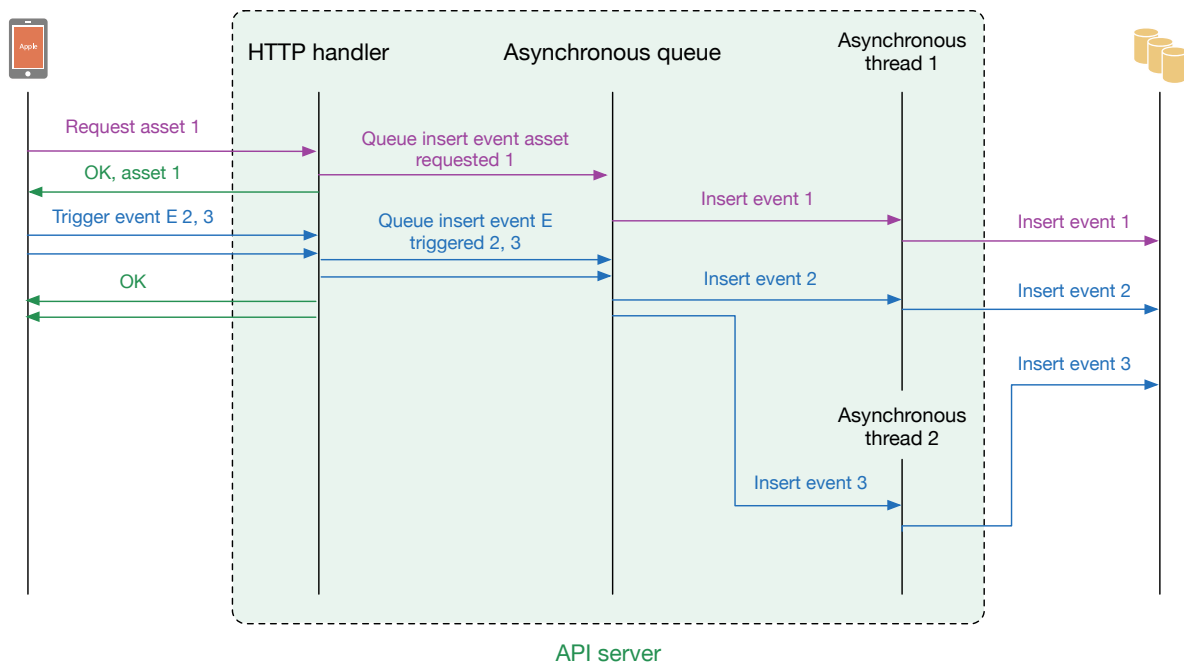


Figure 6 - Original log and event insertion process (simplified)

The HTTP request handler receives the requests from the clients' devices and performs the required operations. In this example, the request targets an asset. The handler will check the request (session key validity, permissions, etc.), fetch the requested asset and send it back to the client. Those operations are not shown on the diagram. However, the HTTP request handler will trigger an event with all the required information about the query for it to be stored in the logging database. It will defer to a virtual asynchronous queue the task of writing the event in the database. The queue, or pool of asynchronous threads, is a Spring-generated system that will distribute tasks across the threads it has to eventually write the events in the database [17].

The workload is similar for event and log pushing by clients' applications. Note that the requests return OK despite the fact that nothing has been effectively written to the database.

Note also that all the asynchronous operations, while being done asynchronously, are still part of the API server. It takes care of the HTTP requests, the read operations in the database as well as the deferred write operations. Therefore, these threads consume CPU and memory, as well as network bandwidth between the database and the API. Moreover, the threads insert only one record each time they are activated, thus being really inefficient. Finally, remember that the load is split across two API servers that are exactly the same, and that therefore has the same kind of thread pool.

3.5 Introducing a message-oriented middleware in Appgrid architecture

Based on the way Appgrid currently works and the message-oriented middleware presentation exposed in section 2 , a new architecture can be suggested to improve the way tasks are currently performed.

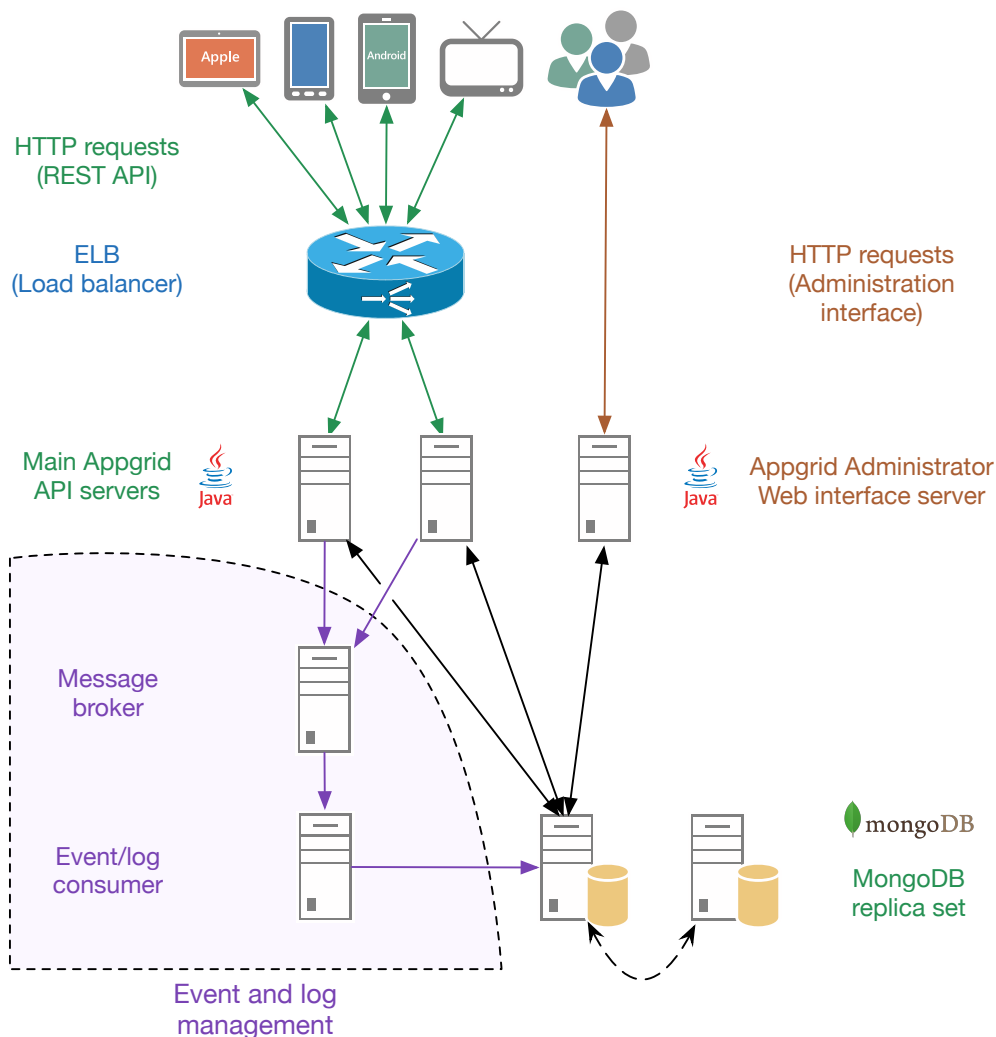


Figure 7 - Message-oriented middleware in Appgrid architecture

A possible architecture is the one depicted on Figure 7. A message broker, hosting various job queues, will receive messages (i.e. logs and events to insert in the database) from the data producers (which are the API servers in this architecture). It will then send these messages to the consumers, which will in turn insert their content into the right MongoDB collection.

The broker may host multiple queues (e.g. one for the events triggered by the application and one for the event triggered by the requests themselves, internally by the API). The consumer may consume data from one or more queue. Messages flow from the API servers to the message broker. In the simplest case, it is a unidirectional communication: messages are sent toward the broker and then forgotten by the API servers¹. The consumer then requests or receives from the broker those messages and processes them at its own pace, inserting them into the database.

3.6 Message-oriented log and event insert process

Considering the architecture described in section 3.5, the event insertion process depicted in section 3.4 can be updated as follow:

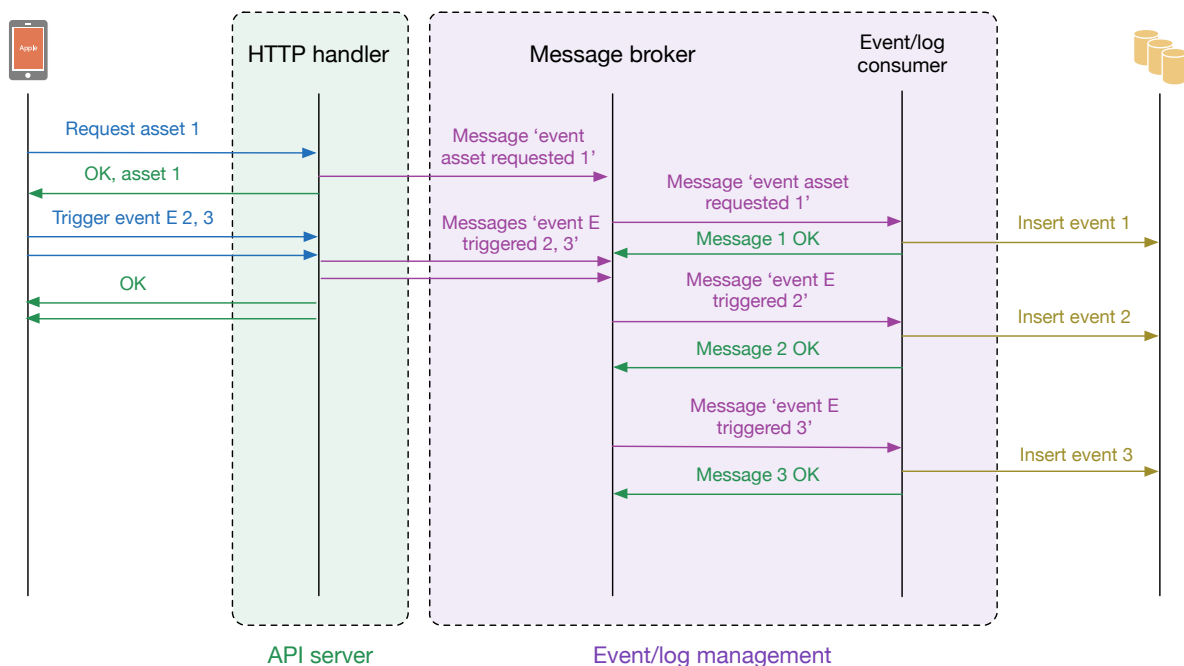


Figure 8 - Basic message-oriented Appgrid log insertion process

The workload proposed in Figure 8 is very basic and not optimized but shows the big picture: the API server is completely relieved from the asynchronous processes and instead defers them to the event/log consumer that it sends messages to via the message broker. The message broker here is set up to feed the consumer on a message-per-message basis, meaning that it would wait for it to confirm that the last message it sent had been consumed properly before sending the next one.

¹ I.e. without publisher confirmations. See glossary page 88.

Though non-optimal, this scheme already has multiple benefits:

- **Process isolation**
The asynchronous insert operations are completely deferred to other instances, and do not interfere with the primary application any longer.
- **Limited resource waste**
While the original process required multiple asynchronous threads to be opened on each API server, none is needed here. One synchronous process (the consumer) inserts events one at a time and confirms it to the broker. It should thus limit the burden on the database, having only one writing connection.
- **Fault tolerance and burst management**
If the API server were to fail, not a single pushed event would be lost, since the queue is now on the message broker, and not the API anymore. This also allows the API to handle way larger traffic bursts without being really impacted (provided the only task a method would do would be to insert an object into a database, which is not the case), leaving all the work they induce to the broker and the consumer.
- **Eased delivery process and operational management**
If the consuming process had to be changed (e.g. adding a pre-processing phase where a field value in the received entities would be changed before being inserted into the database), it could be done online without any data loss. Data would remain queued until the consumer is brought up again. Also, if the traffic gets really important, an operational team may choose to switch or throttle down the consumer, to relieve the database while the system faces a traffic peak.

3.7 Possible improvements to the basic message-oriented log insertion process

Considering the Appgrid environment in which the message-oriented middleware is to be installed, some cleverer solutions can be used to improve the workload presented in section 3.6:

- **Batch inserts**
Since the consumer task is to push data to MongoDB instances, it can take advantage of all the optimized insertion methods the database has to offer. One of them is called batch/bulk insert [18]. It means that instead of sending one record at a time to write in the database, the consumer could send n records to insert in the same collection at once, thus limiting the number of round trips between the database and the server, as well as triggering optimized insertion procedures on the database side.

Instead of passing one document to the *insert()* method in the code, an array of documents can be passed. The problem being populating this array. In the workflow presented in section 3.6, the consumer only had one message to process at a time. Depending on the message-oriented middleware used, this behaviour may be changed, so that the consumer may receive *n* messages without sending a single acknowledgement. This feature is called *prefetch* by some brokers, and can be enforced at the consumer side by others.

By combining *n* message deliveries with a local queue on the consumer side, that would fill up to a certain point, and get flushed to the database in a single bulk insert when the limit is reached, or after a certain period of time (to prevent data retention in a consumer process if it does not receive enough messages), global write throughput is likely to get really better.

- **Batch message sending**

Either from the producer to the broker or from the broker to the consumers, messages are in section 3.6 sent one by one to their destination. This may be fairly inefficient and the same sort of short-term caching and batch sending can be used on both the producer and the broker to limit exchanges between peers. This may not be possible with all the message-oriented middleware available on the market, and may have to be simulated by own code if wanted.

A cleverer architecture could then be built using a process similar to the one presented in Figure 9.

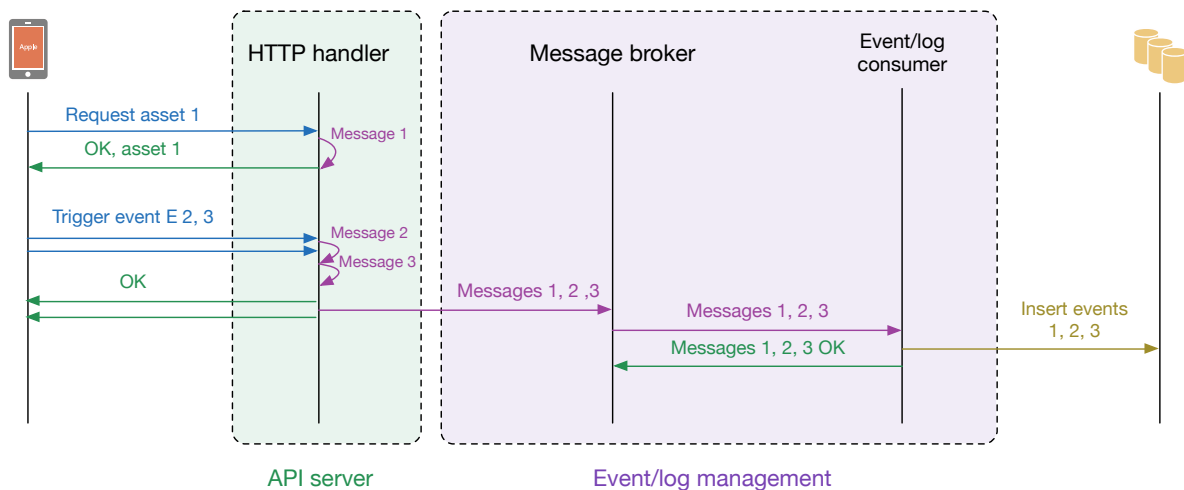


Figure 9 - Optimized message-oriented Appgrid log insertion process

4 Choice of a message broker

The task the broker would have to achieve being defined, it is now time to get an exhaustive idea about which message-oriented middleware exist on the market, and what are their main features. This section presents the state of the art in the domain, and then focuses on two of the main competitors, Apache Kafka and RabbitMQ.

4.1 State of the art in message-oriented middleware

As a new way to think process-to-process communication and to solve the scalability and performance problems that Remote Procedure Call (RPC) involves, several message-oriented middleware papers were published in the 90s [19], [20]. At that time, IBM was the most serious competitor in the field with its MQSeries products, along with some other companies like NEON (New Era of Network) [21]. IBM's products still exist, rebranded as WebSphere MQ, but are commercial, proprietary, and they are therefore not open source.

Other tools appeared in the 00s: the Apache foundation originated various projects, like ActiveMQ in 2004 and Qpid in 2005 [22], [23]. Still supported as of 2014, ActiveMQ is said to be “the most popular and powerful open source messaging” server, according to the project's website. Both ActiveMQ and Qpid also supports JMS, or Java Message Service, which is a Java API that allows Java application to connect to any broker that would implement it, without actually defining anything but interfaces to be implemented by JMS-compliant message broker systems [12, Sec. 5.7].

The message-oriented middleware landscape changed with the initiation of an open standard, Advanced Message Queuing Protocol (AMQP) in 2003 [24]. Originated at JPMorgan Chase in London, in an effort toward a simplified interoperability between financial market operators, it became a global industry work to which 23 companies did contribute (including Barclays, Cisco Systems, Microsoft Corporation and VMware). AMQP defines a standard way to build message-oriented middleware (e.g. in terms of acknowledgement support, push and pull message delivery, etc.), in a programming language- and architecture-neutral way [25]. The final 1.0 version was eventually released in October 2011.



Figure 10 - Logo of various message-oriented middleware or providers (trademarks)

Qpid was the first implementation of AMQP, while RabbitMQ (created by Rabbit Technologies) arrived in 2007 implementing the same protocol, but using *Erlang* instead of Java and C++ as programming language. As of today, “RabbitMQ is used by everyone from small Silicon Valley start-ups to some of the largest names on the Internet” [26, Sec. 1.3]. It has a strong community support behind it, completely supports AMQP, and can take advantage of the capabilities that *Erlang* provides to ease clustering and load distributing operations. It is also open source, seems fast according to some benchmarks, and is used by big companies like Google, VMware and Mozilla [27]–[31]. For these reasons, RabbitMQ was chosen as one of the message-middleware to test in this study.

AMQP is however not the only way to go. Other protocols like STOMP (Streaming/Simple Text Oriented Messaging Protocol) and MQTT (MQ Telemetry Transport) also aims at standardizing exchanges between processes [32], [33]. Therefore, other message-oriented middleware, maybe less popular, also exist, implementing some of them. Qpid provides an implementation of STOMP, Apache Apollo (which is seen as the future of ActiveMQ, and that supports all the three messaging standards mentioned above) [34]... Some quick benchmarks tend to show that they are though not as efficient as AMQP-based middleware [35], [36]. Mentioning ZeroMQ in this section is a bit off-topic, as it is not a complete message-broker system but rather a library that allows one to build its own messaging system [37, Ch. 1]. It was therefore not considered as an applicable solution for the problem presented in section 3.

One last *typology* of message-oriented middleware gathers persistence-optimized systems. Apache Kafka and HornetQ are two message brokers that are specifically designed to be efficient when writing queue data to the disk [38], [39]. They rely on clever page caching methods to avoid large performance hits due to the important number of small write operations that such a system would have to perform when receiving thousands of messages each second. Apache Kafka is a recent piece of software, originated at LinkedIn, which was then made open source and is now maintained by the Apache software foundation. Since the possibilities offered by a long-term message storage scheme may be interesting for some applications – including the one studied in section 3 – and that some benchmarks shown that Apache Kafka was performing well, it was chosen to be the second competitor to be part of this study [40].

4.2 Apache Kafka

Apache Kafka [38] is an open source message-broker middleware, which relies on Apache ZooKeeper [41], both being maintained by the Apache Software Foundation, while it originated at LinkedIn in 2011 [42]. Designed with performance and persistence in mind, Apache claims that “a single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients” [38]. Developed in Scala (an object-oriented and functional programming language that is compiled as Java bytecode and run in typical Java Virtual Machines [43]), it uses a specific vocabulary sometimes that is described in the glossary page 88. The reader is strongly encouraged to refer to the glossary when the concepts do not seem clear in the mainstream document.

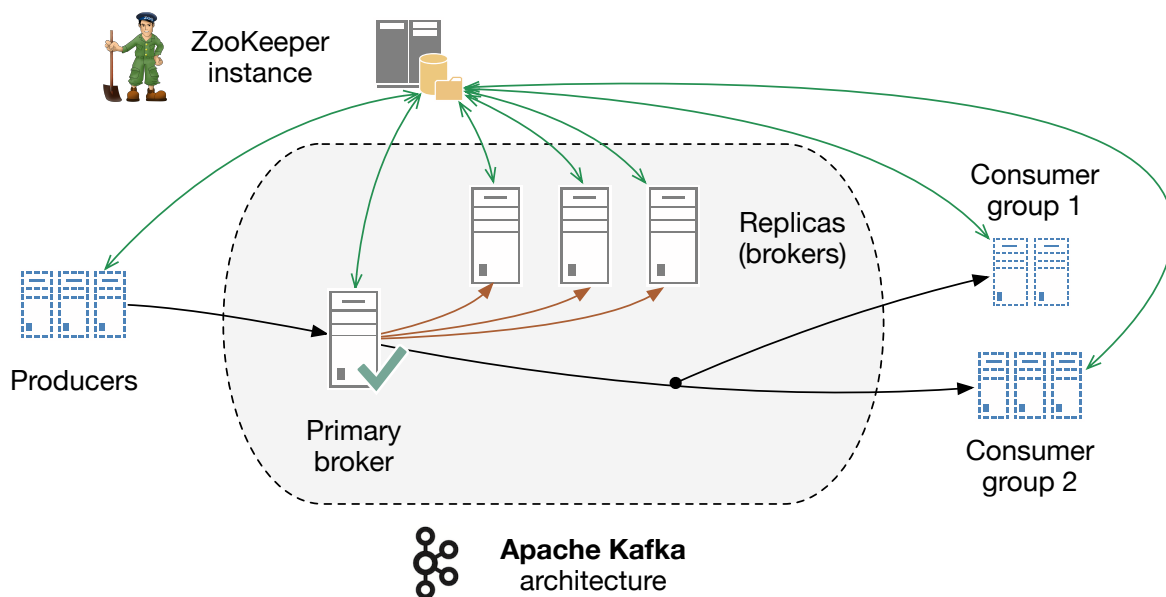


Figure 11 – Global Apache Kafka architecture (with one topic, one partition, replication factor 4)

Kafka is special in the fact that it writes down all the messages to the disk as soon as it receives it. In fact, no messages are forwarded if they were not persisted to the disk (not exactly true though: messages are said to be persisted once they are in the kernel’s *pagecache* [44, Sec. 4.2.]). Therefore: provided that the primary broker received the message, it should not be possible for it to get lost, even if there is an important failure at the broker. Moreover, brokers can have replicas, and a message can be optionally considered as enqueued only once it has also been replicated to all the replicas in the group. This persistence-by-design architecture provides the maximal security level one could expect against software crashes, and should not hurt performance compared to in-memory queues according to the documentation [44].

Note that Kafka is not self-sufficient: it depends on an external Apache ZooKeeper infrastructure that acts as a scheduler and provides a shared database (namely, a hierarchical key/value tree) that all the entities involved in the exchanges use to get and modify the state of the system (e.g. producers will pull from the ZooKeeper instance the address of the primary broker for a topic, consumer groups will store the offset of the last message they read in the tree, etc.). As a consequence, Apache Kafka is not available as a fully bundled application, as it needs a ZooKeeper instance running (with proper, dedicated redundancy) as well. An out-dated version of ZooKeeper is provided in Kafka's package though.

4.2.1 Overall structure

Kafka has like the other message-oriented middleware the concepts of data producers and consumers. Producers push messages to broker, while consumers will process them. However, it also introduces new concepts that it does not share with others:

- **Topics and partitions**

A topic may be seen as an equivalent for a queue in the architecture described in section 2.1. It is the largest message grouping entity. Though, messages are not only pushed to a specific topic by producers, but also to a specific partition. A partition is a subdivision of a topic, and one broker has the responsibility of $\{0 \dots N\}$ partitions, N being the number of partitions in which a topic is subdivided (see Figure 12).

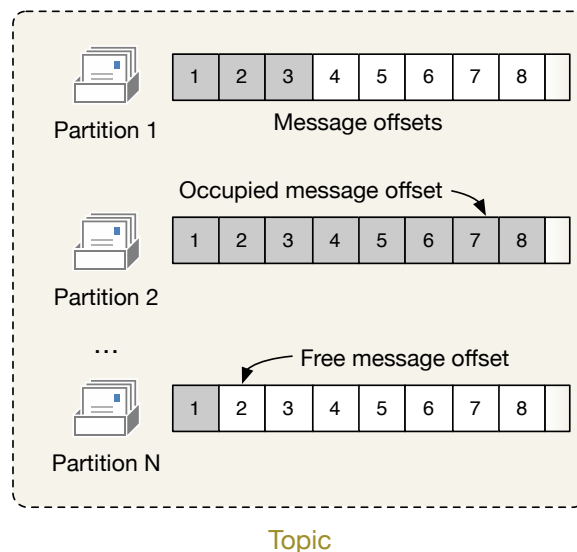


Figure 12 - Architecture of a topic

Each partition has its own message offsets' count. It means that messages can be pushed unevenly to partitions inside a topic. However, since partitions are the smallest entity that a broker can be responsible for (and as such, the smallest replicated entity), it may not be wise. In typical cases, a round-robin partition selection is made. Note that the choice of which partition should a message be delivered to is up to the producer that sends the message.

- **Message offset**

Each message that is stored in a partition is given a unique, incremental integer identifier, as depicted on Figure 12. As a consequence, messages queued in a partition are stored (and thus consumed) in the order they were sent. Having a unique and incremental identifier also enables consumers to “rewind” and replay messages with past message offsets.

- **Consumer groups**

Kafka provides a clever way to both spread the load of consumers across different instances while also allowing simultaneous consumption of the same messages. It introduces for this purpose the concept of consumer group. A consumer group is a group of consumers that do the same task on the same topics and partitions. As such, only one of the consumers a group consists of should receive a copy of a message. Other consumer groups listening for messages on the same topic will also receive a copy of the message (only one per consumer group). Note that the last read message offset for a partition is kept on a per-consumer-group basis, which allows consumer groups not to run at the same time and pace. This consumer group concept therefore gives Kafka support for publish/subscribe message delivery (i.e. consumers register themselves to receive each one a copy of the same message. Creating different consumer groups achieve this mode of operation) and simple queue delivery (i.e. one message is only processed once. This mode of operation is achieved when only one consumer group processes the contents of a topic).

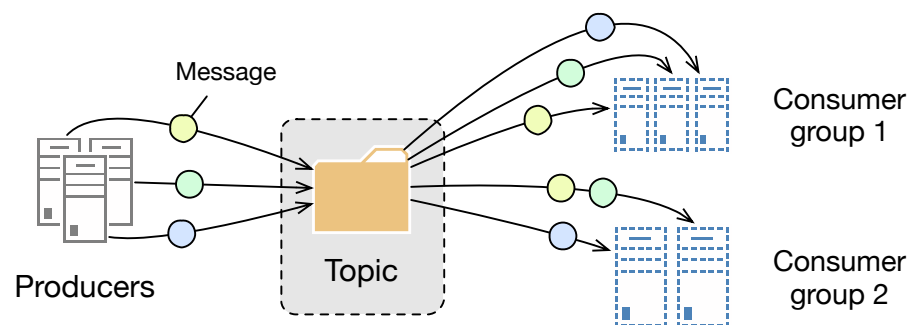


Figure 13 - Consumer group concept (example with one topic and one partition)

4.2.2 Main features

- **Message and queue persistence**

The strongest advantage of Apache Kafka, compared with most of the other messaging systems available is that it enforces message persistence as the rule: it does it by design and one cannot choose to work only with in-memory queues. It is therefore optimized for disk-intensive I/O operations [40]. Note that this message persistence also implies queue persistence, meaning that queues are defined in advance by the broker administrator and will exist until the administrator explicitly deletes them.

- **Per-partition load balancing and replication**

Kafka offers an easy and convenient way to perform load balancing and improve fault tolerance at the broker level: each broker is responsible for some partitions, and will keep a copy of them. The example on Figure 14 shows how a 3-broker system with 3 partitions would be able to split the load across the different brokers. Note that broker 3 does not receive any messages from the producers (and would not send any message to the consumers): only the leader can send and receive messages. It is its duty to keep its replicas in-sync afterwards. Note also that a broker may be the leading broker for none, one or more partitions it is responsible for. The partition distribution across the brokers is done automatically by default, but can be also given manually when the topics are created.

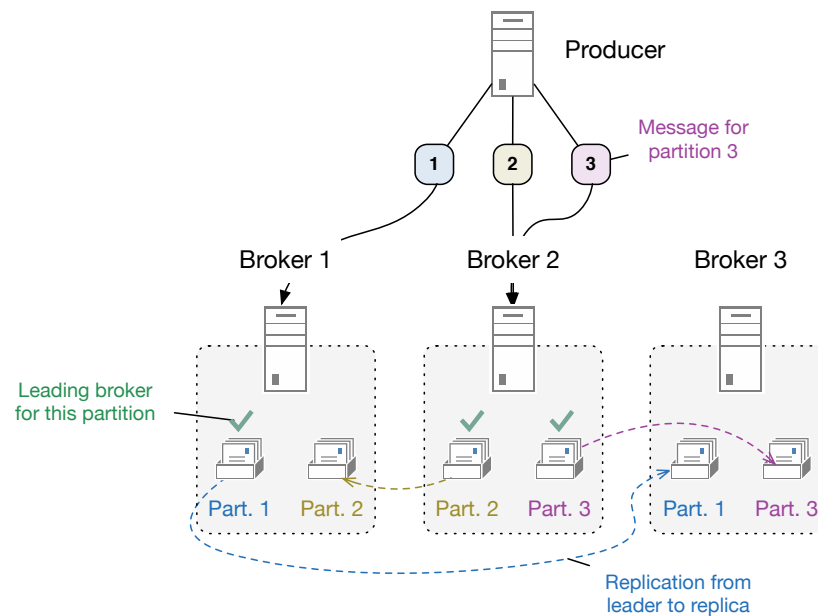


Figure 14 - Replication example (replication factor of 2, 3 partitions, 1 topic)

- **Asynchronous producer**

Contrary to some other message-oriented middleware, Apache Kafka provides a convenient and built-in way to let producers post messages asynchronously toward the broker. This is the behaviour that is explained in section 3.7 and referred to as *Batch message sending*. Producers have a local queue in which they post messages, and flush this queue to the broker when a certain number messages has been queued, or after a certain time. It optimizes network and global resources' utilization.

Asynchronous producers require additional parameters to be set (and possibly tweaked) for them to work. They include:

- A queue timeout: if reached, the queue is flushed to the broker no matter how many messages are in there.
 - A queue maximum size: the maximum number of messages that can be queued without being sent to the broker.
 - A batch size: the maximum number of messages that can be sent in a single operation to the broker.
 - An action to perform if a new message comes while the queue is full: drop it, wait some time to see if a space becomes available or block the thread until the message can be added to the local queue.
- **Enqueuing acknowledgements**
Producers can require the broker to send acknowledgements once messages have been successfully queued (also known as publisher confirmations). They may choose not to, but then take the risk of losing messages (e.g. due to network problems). Multiple acknowledgements schemes are given, among them a simple acknowledgement, sent when the messages is received and enqueued by the leading broker, and a “commit” acknowledgement, sent only when the leader and all its replicas have a copy of the message.
 - **High level consumer API**
Kafka provides two APIs that can be used to build consumer processes: a low and a high-level API. The difference is in the number of tasks that is left to the programmer regarding Kafka’s management. The high-level API takes care of the offset that the application consumes automatically (i.e. it stores it in ZooKeeper once the message is delivered to the logic of the application), it detects and connects automatically to the primary broker of a specific partition, and it is able to detect leaders’ changes in a replica set [45]. It therefore relieves developers from various tedious tasks, but also prevents him from doing more advanced tasks, such as replaying messages or reading only specific message offsets. For these applications, the low-level API and its higher management burden should be used.
 - **Log compaction and compression**
As Kafka stores everything to the disk, the amount of data it keeps will constantly increase and may reach very high values. Keeping all the messages may not be necessary, and log (synonym of topics and partitions) may be cleaned to claim space back. Two modes are available: a delete mode or compact mode. Delete mode is simple: after a given period of time, or when the topic reaches either a defined Time-to-Live or a size limit, the record is deleted. Compact mode is used to clean the log by keeping only the last record of interest (e.g. for a topic storing key/value messages, keep only the last messages with distinct key values).

If the topics are to store large messages, it can be a good idea to enable data compression. Kafka provides built-in settings that allow compression using either *gzip* or *Snappy* compression algorithm [46].

4.2.3 Set-up and management

Apache Kafka is clearly not the easiest and cleanest package one can install and run on a server. Multiple instances (ZooKeeper, brokers) have to be launched to get it up and running, and testing it on one's local machine will quickly require a large screen to watch everything simultaneously. Appendix A — page 101 gathers some information that would help the reader in running Apache Kafka on its own machine, as well as performing the usual administration tasks.

Apache Kafka is programmed in Scala and therefore requires it to work. Packaged versions are available for most architectures and operating systems. A Java Virtual Machine is required as well.

All management operations are done via shell scripts that in turn call Scala classes. As a result, most of the command invocations have a latency that may be a bit bothering while setting up and administrating the instances. The lack of a proper management console makes it quite difficult to be sure that everything is going on well in the system while testing.

Kafka's brokers, as well as consumer and producer clients feature multiple sensors that can help one to keep a look on the overall activity in the architecture. These metrics (e.g. incoming message rate, byte rate, transmission error count) are available as MBeans. They can thus be monitored through any JMX-compatible console. Kafka relies on the Yammer Metrics library to populate those MBeans, which gives, beyond the instantaneous value, some statistics about the past values (moving average, sum, etc.) [44, Sec. 6.6.], [47]. However, no history about the values is provided, which requires a monitoring system to watch the instances 24/7 to have a proper picture of the system.

4.3 RabbitMQ

RabbitMQ is an open source message broker middleware created in 2007 and that is now managed by GoPivotal [48], [49]. It follows the AMQP 0-9-1 (Advanced Message Queuing Protocol)



protocol, which is a standardized and open protocol that “defines the behaviour of the messaging provider and client as well as the broker component in order to achieve interoperability” [25], [50]. RabbitMQ is programmed in *Erlang*. Once again, this section might use technical and unusual words that are specific to RabbitMQ. Those words are likely to be explained in the glossary page 88. The reader is strongly encouraged to lookup for words there if the concepts presented here are not clear enough.

Most of the operations are performed in memory. RabbitMQ is not “disk-oriented”: messages are received by brokers via an exchange (i.e. a logical entry point that will decide based on some criteria in which queue(s) the broker should place a message) and then pushed to the registered consumers. The broker pushes *randomly* queued messages toward the consumers. They thus receive unordered messages, and do not need to remember anything about the queue state (as messages are unordered and pushed by the brokers. They do not and cannot fetch specific messages on their own). Messages are paged out to disc only if there is no more memory available, or if they are explicitly told to be stored.

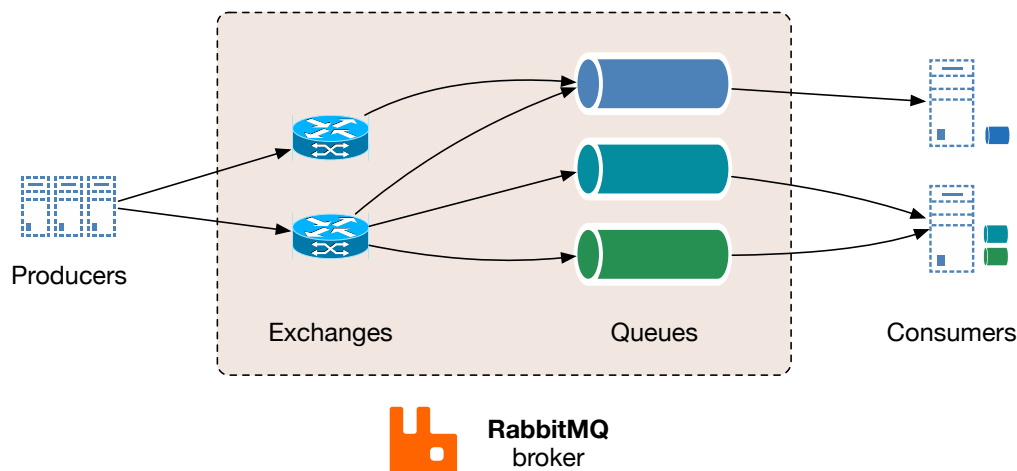


Figure 15 – Simplified overall RabbitMQ architecture

As presented on Figure 15, RabbitMQ features producers that generate messages and send them to an exchange. Exchanges apply routing rules on the message (possibly based on a routing key that the producer put in the message header) to decide whether the message should be delivered to zero, one or more queues (it then duplicates the message). Consumers have a permanent connection with the broker, which therefore knows which consumers are available, and to which queues they did subscribe. The broker pushes messages to the consumers whenever possible (that is, until the *prefetch* count is reached, or the consumer refuses messages).

One main difference with Apache Kafka (presented in section 4.1) is that messages (as well as queues and exchanges) are not persistent by default. All the elements that the brokers manage do not survive (with the default settings) if the broker is restarted or fails. Everything is kept in memory, which is a fundamental difference with the way Kafka works. Fortunately, RabbitMQ offers convenient settings to make both queues and messages durable. Messages themselves can be tagged as durable or not by the producers, so that they can choose on a per-message basis not to persist some non-critical messages sent to a durable queue.

4.3.1 Overall structure

- **Exchanges and bindings**

Constitutive parts of the AMQP protocol, exchanges and bindings dispatch messages that come from the producer to queues [51, Sec. 5.11]. A producer sends a message to a specific exchange (and not a queue), which in turn will decide whether it should redirect, duplicate or discard the message. Different types of exchanges are standardized and RabbitMQ supports all of them [52].

These routing choices can be altered by the value of the message's routing key, which is put in the headers by the producers. The simplest example of exchange is the *direct exchange*, as depicted on Figure 16:

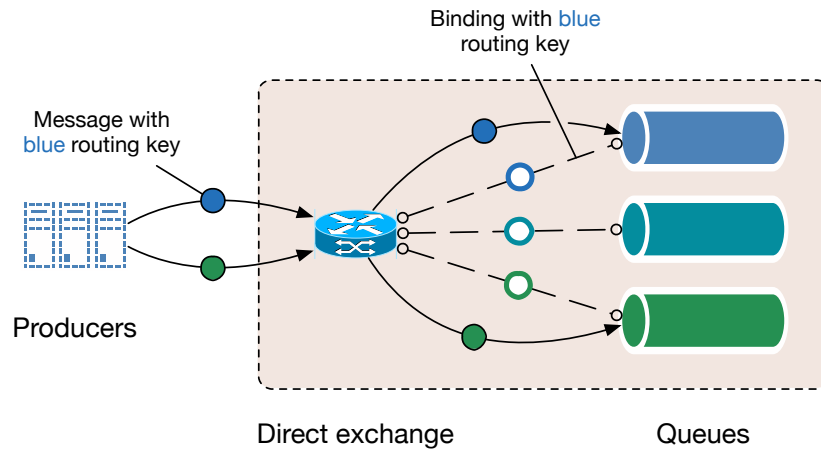


Figure 16 - Direct exchange example (routing keys = queue names = colours)

It namely routes messages via the binding that has the same routing key as them. In the example on Figure 16, the presented exchange is also the default exchange, which is an exchange that producers can send message to by specifying an empty string as destination exchange. The default exchange binds automatically all the queues that are created in the system to itself, with a routing key equals to the name of the queue. Therefore, sending a message to the default exchange with the routing key "appgrid" will route the message to the queue named "appgrid", if it exists.

Another interesting exchange type is the *fan-out* exchange. As shown on Figure 17, it will simply duplicate the messages it receives and put them in all the queues to which the exchange is bound. This is specifically useful to achieve publish/subscribe message distribution scheme, where multiple consumers should perform different tasks on the same messages. One and only one consumer listening to a queue in RabbitMQ consumes a message put in this queue. Duplicating messages is thus the only way to have the message processed multiple times by different consumers.

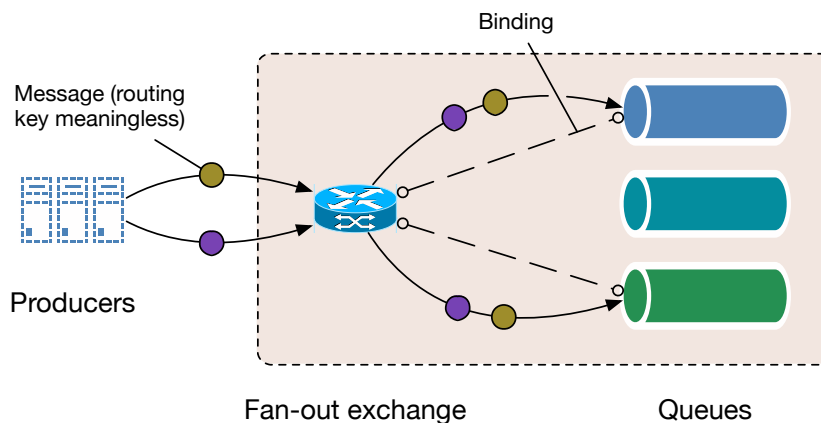


Figure 17 - Fan-out exchange example (routing key meaningless, three queues, two bindings only)

More complex routing can be achieved using other exchange types (*headers*, and *topic exchanges*), which will not be explained in this document. The reader may find additional information in the references and the glossary [52].

- **Queues**

Queues in RabbitMQ are the endpoints for all messages that were successfully routed (i.e. that were not dropped by an exchange). A message in a queue will be (ideally) delivered only once, to only one consumer that subscribed to the queue. This design implies that to mimic the consumer group concept (see section 4.2.1) that Kafka features, messages must be duplicated, while each “false” consumer group should be bound to one queue.

- **Channels**

Channels are a low level component of AMQP, implemented in RabbitMQ, which is used to multiplex connections from the same producer or consumer to the same broker but that are used by different threads for instance. Instead of having multiple TCP connections established between the peer and the broker, only one is instantiated and kept active, and RabbitMQ performs multiplexing to have all the channels share this single TCP connection.

4.3.2 Main features

RabbitMQ is a pretty standard message queuing system, and shares most of the general concepts presented in section 2.1. However, even though it follows the AMQP standard (which in itself features some “unusual” functionalities), the development team developed some extensions to the protocol, extensions that can be useful in the studied use case:

- **In-memory storage and optional (limited) durability**

As already mentioned in the previous section, RabbitMQ stores all the elements it deals with in memory, while providing flags that can be put on queues and messages to mark them durable. However, the achieved durability is not comparable to the one that Apache Kafka features, in that the durability provided is just temporary: messages are written down to the disk only if there is no consumer that can process the message right away. Messages (no matter their *durable* flag value) can also be paged out to the disk if the memory gets exhausted on the broker. Once the message has been delivered (or that its consumption has been confirmed if these confirmations are enabled), the message is forgotten, even if it was tagged as *durable*. RabbitMQ does not provide a way to enforce permanent storage in a similar fashion to what Apache Kafka does, and as such does not allow consumers to replay old messages.

- **Reliable delivery and message rejection**

Messages delivered to consumers can be required to be acknowledged: this ensures that messages have been successfully consumed before they are removed from the broker's memory. By default, no acknowledgements are needed, and a successful delivery to one of the registered consumers of a queue is enough for the broker to forget about the message. Consumption confirmations help reducing the number of unprocessed messages, while also giving the opportunity to consumer processes to notify the broker that a message could not be processed. They may indeed reject a message, and potentially queue it again for redelivery. For some use cases (e.g. a database that becomes temporarily unavailable and therefore prevents a message's content to be written down to it, while the connection with RabbitMQ does not encounter any failure), this might be a convenient feature.

- **Dynamically declared queues and exchanges**

While the administrator of the broker can create them in advance, new queues and exchanges can be created directly into the code of producers and consumers. Actually, this is a very good practice to simulate their creation to ensure that messages will not be pushed to inexistent queues and exchanges [53].

- **Permission management and SSL support**

RabbitMQ requires producers, consumers and administrators to authenticate before they can do any operation on the broker. Different permissions can be given to different users, thus allowing the enforcement of an efficient and fine-grained access control. SSL is also supported and can be used to authenticate and encrypt data transmitted over the channels between the brokers and the producers and consumers.

- **Multiple load balancing and replication alternatives**

Thanks to the distribution capabilities an application coded in Erlang offers, RabbitMQ features different ways to replicate configurations and queues' contents. One of them, as depicted on Figure 18, is to create a RabbitMQ cluster:

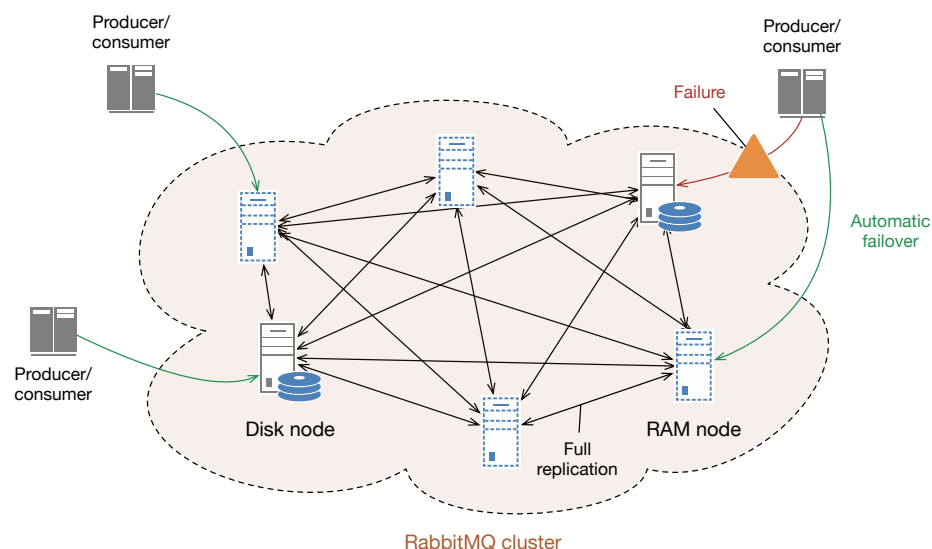


Figure 18 - RabbitMQ clustering with queue mirroring policy set to *all*

In a cluster, individual nodes can fail, as there is a permanent full replication between all the members. Two types of nodes, disk and RAM nodes, make the cluster. A disk node writes its content to disk (exchange, queues and their contents, etc., while a RAM node keeps everything in memory (except for the queue contents, if they are explicitly told to be durable or too large). RAM nodes are usually faster then, and disk nodes are here to allow cluster recovery in case of complete shutdown (at least one is required). Peers can push and pull data from any of the nodes, at any time, provided that the node to which messages were pushed is still up and running (i.e. messages are stored only on the node that was the destination of the message. Messages are not replicated across nodes by default). Please note however that the links between the nodes that make the cluster must be highly reliable with low latencies for it to work properly, and that a bidirectional connection must be maintained between all the nodes [54].

The fact that messages are not replicated across the replica set makes the cluster quite useless. But thanks to the mirroring capabilities featured by RabbitMQ's policies, queue contents can now be replicated across either none, a certain fixed number or all the members of a replica set, and provide automatic master election and failover [55]. New nodes can be added on the fly, either catching up naturally (no pre-provisioning: as the queues are emptied on the old nodes, new nodes will eventually have the same contents as the other nodes) or being provisioned with the queue data before the node is put into function. This provisioning however may imply some unavailability time, as the node should have the exact same contents as the others. Note also that this mirroring feature does not do load balancing by itself: peers will publish and consume messages from the master node only, the replicas being used only in case of failure.

RabbitMQ also provides two other alternatives to share the load across multiple brokers. Namely, federation and shovel can be used on unreliable network links to share or copy messages from one broker to another. Messages can be buffered at the entry point in case of failure of the network link, and sent when it goes up again. Those alternatives allow for a partial load balancing approach: one may federate only some of the queues, and keep others local. More information on these approaches can be found in the references [56].

4.3.3 Set-up and management

RabbitMQ is available for multiple operating systems and platforms, quite often as a packaged application. Nicely integrated and running as usual system services (therefore manageable through typical *service* command line invocations), it can be started right away after its installation. The default settings are usually enough and can be used without any problem. The reader might find Appendix B — page 104 useful to test and manage a RabbitMQ installation.

RabbitMQ comes bundled with different plugins, one of them being dedicated to management and monitoring purposes [57]. Very easy to set up, it gives administrators a nice and convenient way to manage a broker, via an all-in-one web user interface. Queues, exchanges and channels can be watched and configured using the interface. Graphs are plotted and the brokers record automatically and continually the values of the different metrics given, so that they are always accessible, with historical data. Very handy when implementing and testing a new architecture, queues' contents can be completely or partially dumped and displayed through the web interface, and can also be completely purged if necessary. Managing the whole system is really easy and is a matter of few mouse clicks.

4.4 General feature comparison between Apache Kafka and RabbitMQ

The following tables summarize and compare the set of features described in sections 4.1 and 4.3 available in Apache Kafka and RabbitMQ:





	 Apache Kafka	 RabbitMQ
Creation year	2011	2007
License	Apache (Open source)	Mozilla Public License (Open source)
Enterprise support available	✗	✓
Programming language	Scala	Erlang
AMQP compliant	✗	✓
Officially supported clients¹ in	Java ²	Java, .NET/C#, Erlang
Other available clients in	~ 10 languages (incl. Python and Node.js)	~ 13 languages (incl. Python, PHP and Node.js)
Documentation, examples and community³	~ Immature and incomplete	✓

Table 1 - General information comparison between Apache Kafka and RabbitMQ

	 Apache Kafka	 RabbitMQ
Broker capabilities		
Main storage space	Disk	RAM
Ordered storage and delivery	✓ At partition level	✗
Queue content persistence	✓ Complete and mandatory	~ Temporary and optional
Message deletion	After reaching size or time limit	Immediately after confirmed consumption
Queue data compression	✓	✗

¹ Used to develop producer and consumer processes.

² Though unclear. See [58].

³ Author's opinion. Details about this point to be given in results part.

Predefined queues on broker	✓	✓
Multiple different consumers of same data set ¹	✓	Needs queue duplication
Load balancing across a set of same consumers	✓	✓
Remote ² queue definition	✗	✓
Advanced/conditional message routing	✗	✓
Permission and access control list support	✗	✓
SSL support	✗	✓
Clustering support	✓	✓
Self-sufficient	✗	✓
Management and monitoring interface	✗	✓
	Message to partition only	
	JMX and CLI-based	Web and CLI-based

Table 2 – Broker capabilities comparison between Apache Kafka and RabbitMQ



Java producer capabilities	 Apache Kafka	 RabbitMQ
Batch delivery	✓	✗
Asynchronous publishing	✓	✗
Publisher confirms (acknowledgements)	✓	✓

Table 3 – Producer (official Java client only) capabilities comparison between Apache Kafka and RabbitMQ



Java consumer capabilities	 Apache Kafka	 RabbitMQ
Batch fetch/delivery ³	✗	✗
Push delivery from broker	✗	✓
Pull delivery (fetch)	✓	✓
Prefetch count ⁴	✗	✓
	Implicit	
Message replay capability	✓	✗
Consumer confirms (acknowledgements)	✗	✓
	Implicit	
Message rejection ⁵	✗	✓

Table 4 – Consumer (official Java client only) capabilities comparison between Apache Kafka and RabbitMQ

¹ I.e. support for publish/subscribe delivery model (see section 4.2.1 page 29 about consumer groups and section 4.3.1 page 34 about *fan-out* exchanges).

² I.e. by consumers and producers.

³ Reception of multiple messages sent in a single batch from the broker.

⁴ Refer to section 3.7 page 24.

⁵ Also known as re-queuing. See section 4.3.2 page 36.

4.5 Preliminary conclusions

Apache Kafka and RabbitMQ are two message-oriented middleware that have a fairly different set of capabilities, the main differentiating point being the way they store and keep (or not) messages. Depending on the expected data flow and the tasks to be performed, one or the other may perform better.

However, from a workflow point of view, they both allow messages to be queued, and tasks to be distributed to one or more consumers. Messages could be written to the disk, and consumers may process them whenever they are up and running, which is the main objective sought in implementing such a message bus: isolating producing and asynchronous consuming processes.

To have a better idea about the performance, the ease of use and the features of these systems, both have been implemented and tested in Appgrid. The following sections describe this implementation.

5 Implementation in Appgrid and test

Apache Kafka and RabbitMQ seem both to have the required features to be used in Appgrid. To be sure that it is the case, both of them were implemented in Appgrid and tested using the same tools to compare their behaviours. This section presents the implementation details and the test infrastructure.

5.1 Motivations and goals

The ultimate goal in implementing a message-oriented middleware in Appgrid is to completely isolate the write operations in the database that are currently performed asynchronously by the main application itself. As described in section 3, the Appgrid API itself and the devices connecting to it can generate events and push logs that need to be written down to the MongoDB database the application relies on. This does not need to be done synchronously, nor has it any critical time constraints regarding the delay between the publishing of the events and the time they get written in the database. Therefore, these write operations can be deferred to another external process (a data consumer) that will perform them in a dedicated context, at its own, throttled pace.

Two MongoDB collections (namely, *EventLog* and *ApiLog*) will be concerned, and the write operations performed by the API asynchronously by the main application right now will be moved to another, completely independent process. More information on the Appgrid architecture in itself and the expected benefits of this update may be found in section 3 page 19.

5.2 Code architecture

Since Appgrid is an all-in-one, bundled Spring Java application, it is not very convenient to make a quick change in the code and test the new version, as the build and deployment times are prohibitive (in the magnitude of 500 seconds). Therefore, hardcoding parameters in the code was not a good idea, even for testing purposes, and a flexible version, configurable via external files, has been developed.

5.2.1 Core Appgrid modifications

Taking advantage of Java's feature like interfaces, the implementation has been developed in a modular way to avoid code duplication, and separate generic code processing from broker-specific operations. The class diagram for the newly created *tv.accedo.appgrid.data.bus* package is presented on Figure 19.

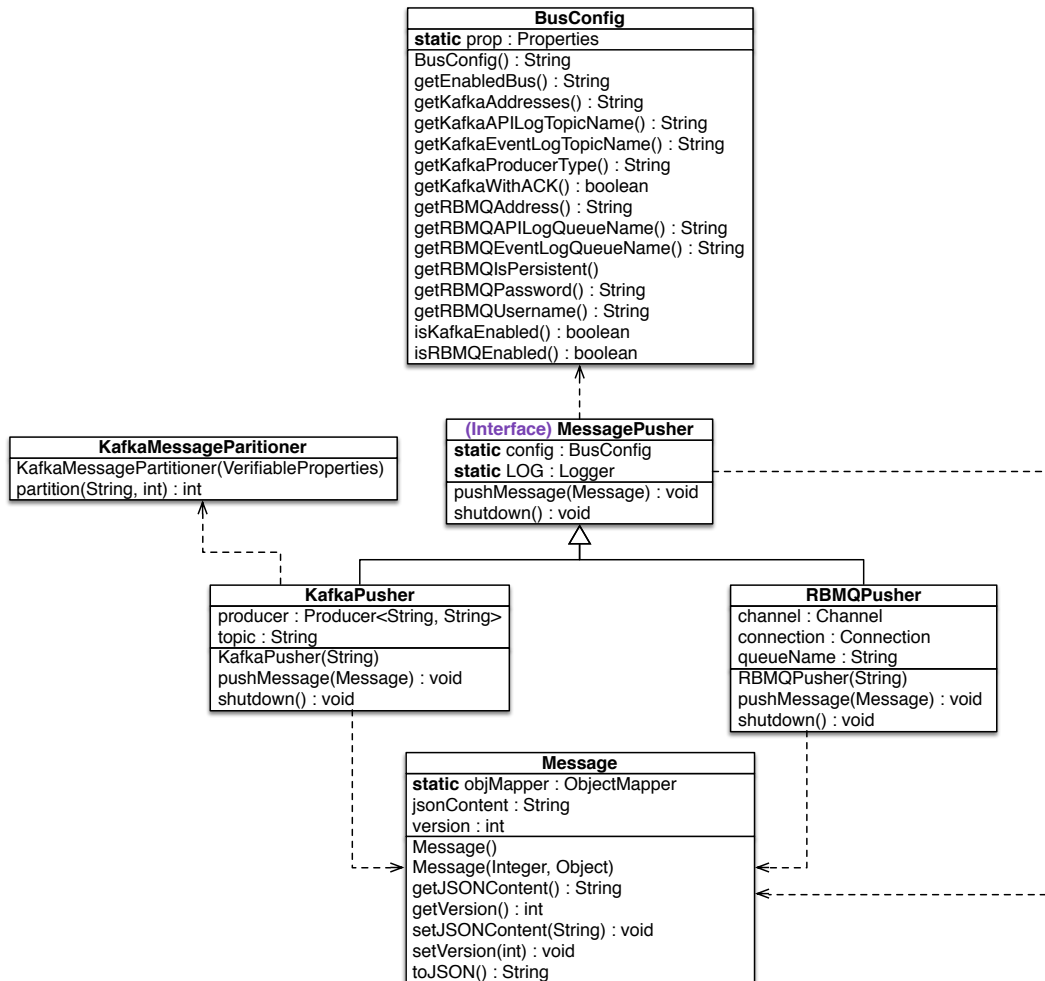


Figure 19 - Class diagram, *tv.accedo.appgrid.data.bus* package

- BusConfig**
 Singleton class used to load and provide to other classes settings read from the configuration file (a default one is hardcoded if none is provided in */conf/accedo-bus.properties*).
- MessagePusher**
 Interface that provides a link to the configuration singleton and defines the two primary methods that all message pushers should implement: *pushMessage(Message)* and *shutdown()*. Wherever a thread needs to open a connection toward a message broker, it should have a reference to a *MessagePusher* object, instantiated as either *KafkaPusher* or *RBMQPusher*.
- KafkaPusher**
 Class that has all the logic to push messages to a Kafka broker. Each instantiation results in the creation of a new TCP connection to the broker that will push messages to the specified topic name and using the *KafkaMessagePartitioner* partition chooser.

- KafkaMessagePartitioner**
 Class used by *KafkaPushers* to decide to which partition of a topic should a message be sent. It achieves random distribution among all the partitions the topic has, no matter the routing key. Whereas Kafka's documentation claims that this behaviour should be obtained with a message sent with a *null* routing key, it was not the case during tests and therefore resulted in this manual implementation [44, Sec. 5.1].
- RBMQPusher**
 Class that has all the logic to push messages to a RabbitMQ broker. Each instantiation results in the creation of a new TCP connection to the broker, together with a new channel dedicated to the queue whose name is given as argument when the object is constructed.
- Message**
 Model class that defines the structure of a message's content. This is a serializable class. Please refer to section 5.3 for more details.

Note that this architecture allows an easy change of broker in the code: where a connection to the broker is needed (that is, in our case, in the method handlers related to the posting of events to the *EventLog* and the *ApiLog* databases), just instantiate an object that implements the *MessagePusher* interface.

5.2.2 Consumer process

The consumer process is organized around the same type of modular architecture, and can consume data from both Apache Kafka and RabbitMQ (at the same time, if needed). It uses a per-queue and per-topic threaded architecture that results in one TCP connection being opened for each queue that the consumer is registering to.

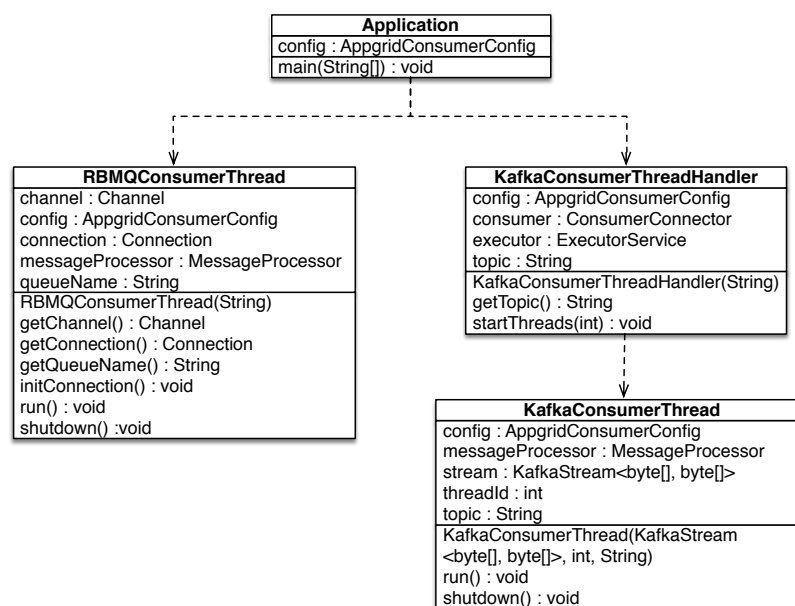


Figure 20 - Thread architecture of consumer process class diagram

Figure 20 presents the main class and the two message listener classes, that will be instantiated and run as separated threads, *RBMQConsumerThread* and *KafkaConsumerThread*. They implement the required logic to connect to the brokers and listen to or fetch messages. *RBMQConsumerThread* implements the push API (it will receive messages directly from the broker, no need for threads to fetch them), while *KafkaConsumerThread* fetches them. *Application's* main method instantiates a thread of each of them depending on their activation state (by reading the */conf/appgrid-bus.properties* configuration file). One thread (per broker) is created for each type of messages that needs to be processed: namely, one to process *EventLog* messages, and one for *ApiLog* messages. They have a reference to a class that inherits from the abstract *MessageProcessor* class.

Figure 21 shows the implementation of the message processing classes (i.e. those in charge of effectively dealing with the content of the messages).

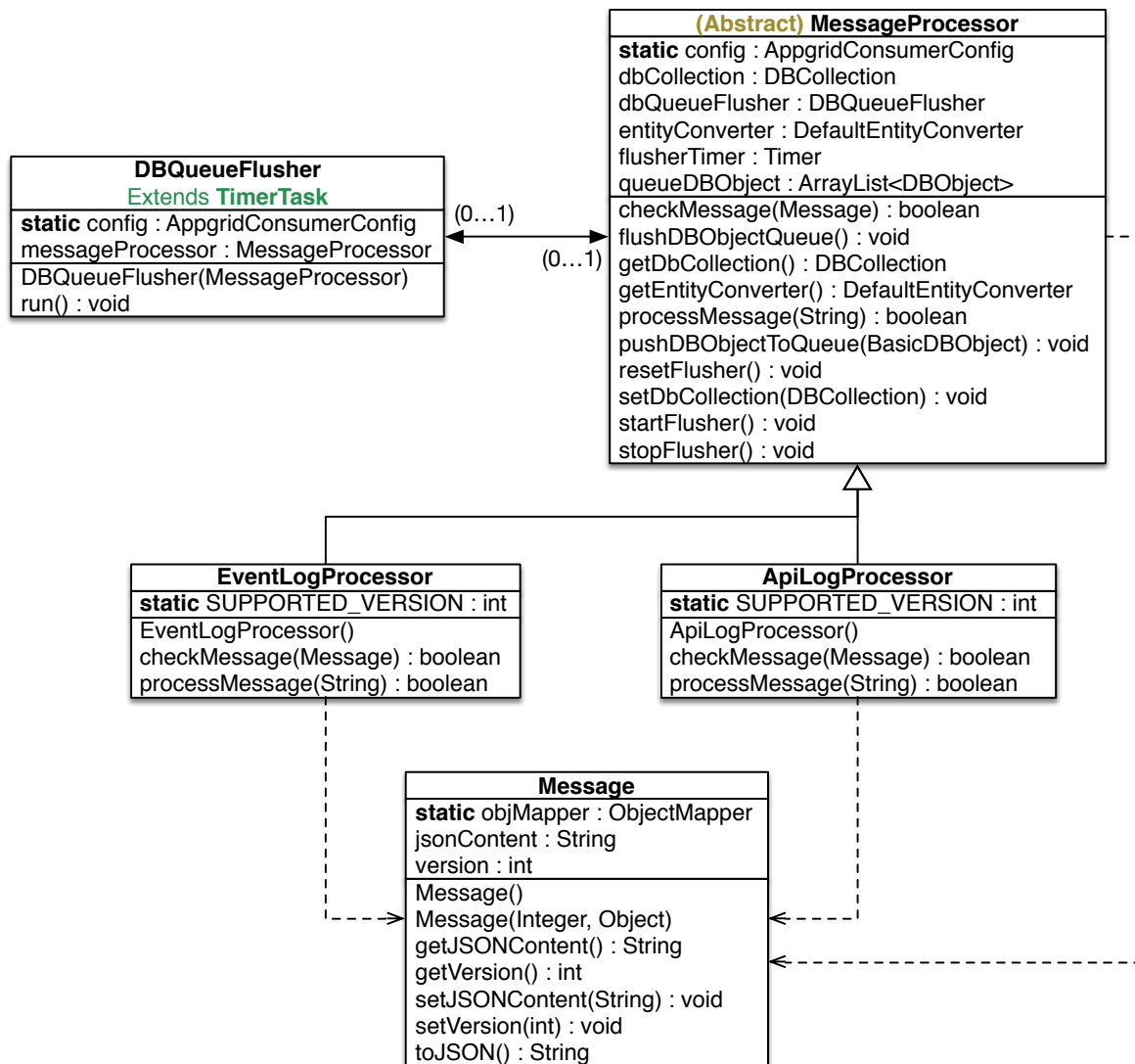


Figure 21 - Message processing classes of consumer process class diagram

- **MessageProcessor**
Abstract class that implements the logic needed to perform actions on the database and has the needed attributes to handle an array (or queue) of *DBObject* as well as a timed flush of this queue to the database. A *DBObject* object is a MongoDB-friendly class that represents a document to be inserted into a MongoDB collection. *MessageProcessor* also provides control over the *DBQueueFlusher* task used to pilot the timed flush of the *DBObject* queue.
- **EventLogProcessor**
Class that extends *MessageProcessor* and that defines the content of the two abstract methods *processMessage(String)* and *checkMessage(Message)*. Takes care of checking, deserializing and processing messages pushed in an *EventLog* queue.
- **ApiLogProcessor**
Same as *EventLogProcessor*, but for *ApiLog* queues.
- **DBQueueFlusher**
Class that extends *TimerTask*, which is used by *MessageProcessor* object. It is the task executed on a regular basis by *MessageProcessor* objects that have had their *startFlusher()* method called. When executed, it just calls the *flushDBObjectQueue()* method of the *MessageProcessor* object it is bound to.
- **Message**
The same class as presented in section 5.2.1.

The way *DBObject*s get written to the MongoDB collection needs some explanation: it uses the bulk/batch insertion capabilities of MongoDB. As presented in section 3.7 page 24, batch insertions mean that instead of sending one document at a time to be inserted into the database, an array of those documents is sent at once. Therefore, *MessageProcessor* objects have an *ArrayList* of *DBObject*s. Whenever a thread requires a document to be written to the database, it is pushed into this array. The methods called to put a new document in the array and to flush the queue are declared as *synchronized*, to prevent simultaneous flushes and documents losses (as two threads can act on the queue).

The queue can be flushed in three cases:

1. **The queue reaches its maximal capacity**
This capacity can be set in the configuration file. By default, it is set to 500 documents. The current number of elements in the queue is checked after each insertion.
2. **The scheduled *DBQueueFlusher* task of the object is executed**
If the automatic *DBQueueFlusher* task were started, the queue would be flushed regularly, at a rate that can be set in the configuration file. Note that the timer is reset each time the queue is flushed (i.e. when the queue is full and then flushed, the timer is reset).

3. The *shutdown()* method of the object is called

It prevents any non-pushed modifications to be lost in case of controlled shutdown.

Note that the consumer process relies on the official and native Java MongoDB client to connect to the database. The *insert()* method is used when flushing the queue, with the *ArrayList* of *DBObject*s as argument. The default settings are used to connect to the MongoDB instance.

A last word about consumer acknowledgements: for RabbitMQ, they can be enabled. In the test configurations used, consumption acknowledgements were sent as soon as the *DBObject* contained in the message was pushed to the queue, and not when the queue was flushed. This may lead to data loss in case of consumer process failure, but may speed up the overall insertion process.

5.3 Message structure and protocol

Message-oriented middleware deal with messages. As already mentioned in section 2.1 page 12, the messages they carry are a string of bytes and nothing more. Depending on the broker, they may be wrapped into a more structured element that may provide some headers. Kafka does not provide such headers, while RabbitMQ has some flags (like the *redelivery* flag in case of rejected delivery. See section 4.3.2 page 36), a routing key and possibly additional custom headers that the consumer can see. Complete freedom is given to developers that use these systems: it is their duty to design a protocol and a standardized structure for the message contents they exchange for having a reliable and sustainable architecture.

The content of the messages depends on the type of operations the consumers need to do. It may be totally fine to just post a raw ASCII string to the queue if the only task the consumer achieves is printing it asynchronously on a printer or a screen. However, in most cases, they will need to perform more advanced tasks. And thus require a more carefully thought design.

The operations that are to be deferred to consumers are database insertions. For this first test, this is the only operation they will have to do. They will receive prepopulated model objects that they just have to insert into the database. Why not serialize these objects using some standard formatter like JSON and push the resulting string to the broker? It would work, but some considerations related to sustainability and architecture evolution should also be taken into consideration:

- **Overhead due to serialization**

If the objects that are to be serialized are small, the overhead due to serialization may exceed the actual size of the data inside the object. Take the example of this simple object, serialized in JSON:

```
{
  "my-first-very-long-boolean-field":"true",
  "the-second-boolean":"false"
}
```

Assume that the application logic that will receive this message knows that the message should contain two Boolean values, in a given order. Why not just send them as "1;0", or even "10" then? Serialization using standardized formatter might not always be a good solution.

- **Collection, table model changes**

While not being very important with MongoDB collections, as documents can be stored in a collection no matter the structure of the previous ones, it may not be a good idea to use model object directly serialized as message content. Assume that the collection structure has changed and that it now stores documents with a *dateTime* field instead of a *date* field. Provided that code in both the producer and consumer side is updated to take into consideration this specific change, it is likely that stale (or rather, data that complies with an old model structure) messages are still in queue. Those messages would cause the consumer to fail, and depending on the implementation of the consumer, may even cause problem in the database.

Wrapping the actual model, data object structure serialization, into an immutable and predefined structure might be a better idea. It may be a very simple one, but would still provide a level of flexibility that would prevent most common errors. Adding an incremental version counter to this structure, together with the real message content, would allow a consumer thread to check whether it is compatible and knows how to deal with this message version or not. Besides, it would be possible to add routines to the consumer so that it would be able to update old data structures to the most recent one (e.g. converting the *date* field to a *dateTime* field with time initialized to midnight). This may be specifically useful with Kafka, as old messages are de facto stored for quite a long time, and may be replayed by consumers.

- **Serialization performance and binary serialization**

Multiple serialization formats exist. JSON, XML, CSV, YAML... the most obvious and used are though human-readable formats. One can take the representation of an object in XML or YAML and still be able to figure out what is the content of the object and its structure. While this can be good for testing, it may not be the most optimized way to encode data sent from a software process to another software process.

Binary serialization formats can come in hand in this case. Relying quite often on a schema or an IDL¹ to define their structure, it is in most cases impossible for a human to read them directly. Instead of a text-based serialization, it generates binary-based representations of the objects that as such are likely to be more compact and efficiently processed. BSON (“Binary JSON” [59]) or Apache Avro (the serialization process used in communication between Hadoop nodes) are examples of binary formats [60].

According to some tests, Apache Avro is more compact (it takes about 3 times less space compared to JSON-encoded objects), and (de)serialization operations are also faster (depending on the JSON library used, operations can be 2-4x faster compared to JSON) [61]. If the number of messages to parse is important, and their size become large, switching to this kind of formatters might improve the overall performance while limiting the waste of disk space, memory and bandwidth.

For these tests, a very simple object structure was used. Figure 22 describes the overall message structure and how its content is converted back and forth to be exchanged via a broker.

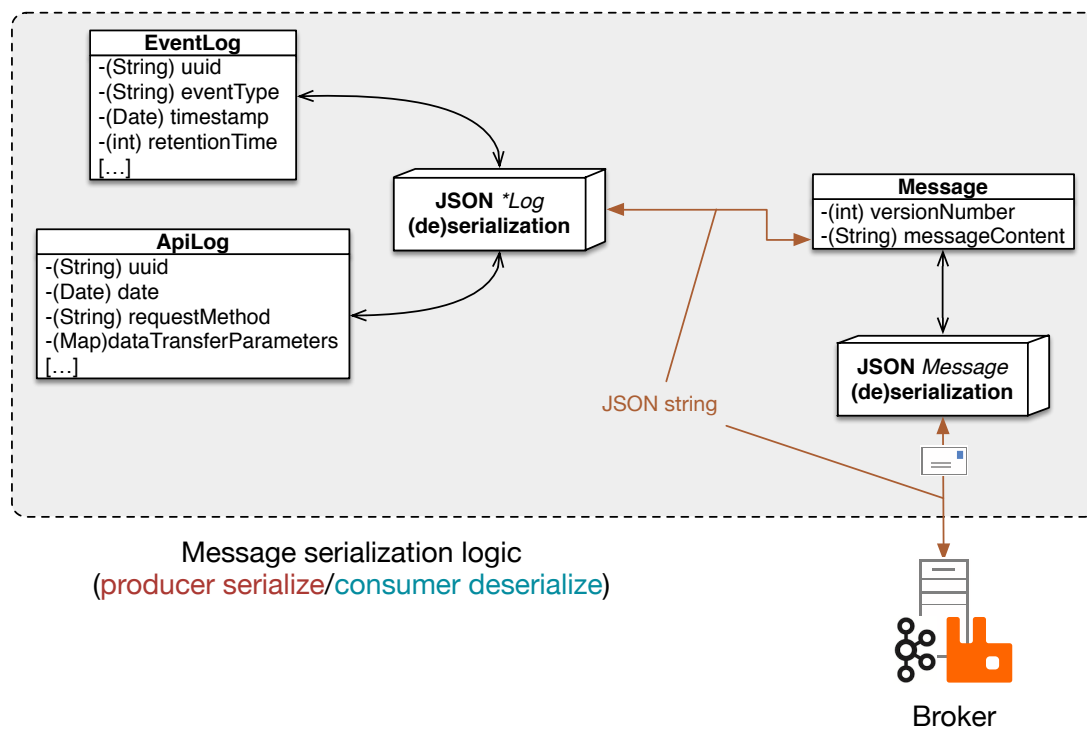


Figure 22 - Message structure and serialization logic

¹ See glossary page 88.

Message, *EventLog* and *ApiLog* are three Java classes, which act as model classes. An *EventLog* or *ApiLog* object is serialized to JSON and the resulting string is put in the *messageContent* attribute of a *Message* object. The *Message* object itself is then serialized to JSON and the resulting string is used as the real message, sent to the broker for delivery. Consumers would reverse the process, but they would have to know which final model class they should use, that is, they need to know whether the message contains an *EventLog* or an *ApiLog* object representation. This differentiation is implicitly done by the queue from which the consumer fetches the data: queues are filled with only one type of messages, thus making it easy for consumers to identify what kind of messages they receive.

5.4 Queue/topic architecture

As mentioned in the goals of this implementation section 5.1 page 42, two MongoDB collections (*ApiLog* and *EventLog*) will be filled by the consumer process. To make management simple, and even though the messages have the same wrapped architecture, two queues (or topics for Kafka) were created, one for each collection. Figure 23 presents the architecture implemented in Apache Kafka for the tests.

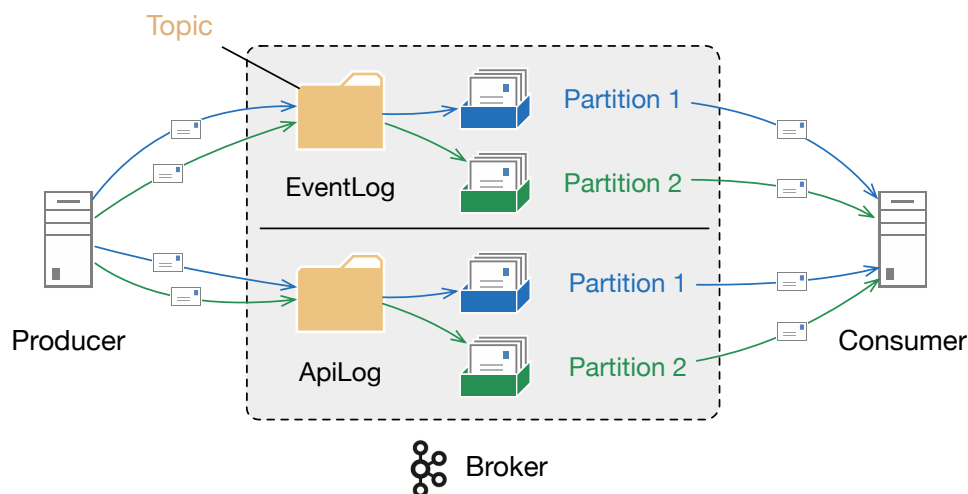


Figure 23 - Kafka topic architecture

Two topics are created, each with two partitions. Producer (i.e. the modified Appgrid API) is coded to randomly put the messages in either one or the other partition (see *KafkaMessagePartitioner* class description section 5.2.1 page 42), so that each partition would on average receive 50% of the total message stream. All the partitions are hosted on a single broker, with a replication factor¹ of 1. This architecture was chosen to take advantage of parallelism capabilities of Kafka: each partition can be filled and read independently, so the consumer process will in fact have 4 consuming threads, one for each partition of the 2 topics. This is to comply with the specific guidelines that should be followed when designing a consumer using the high level API which consumes data from multiple partitions [62].

¹ See glossary page 88 and section 4.2.2 page 30.

The test architecture to accommodate the two message streams with RabbitMQ is shown on Figure 24.

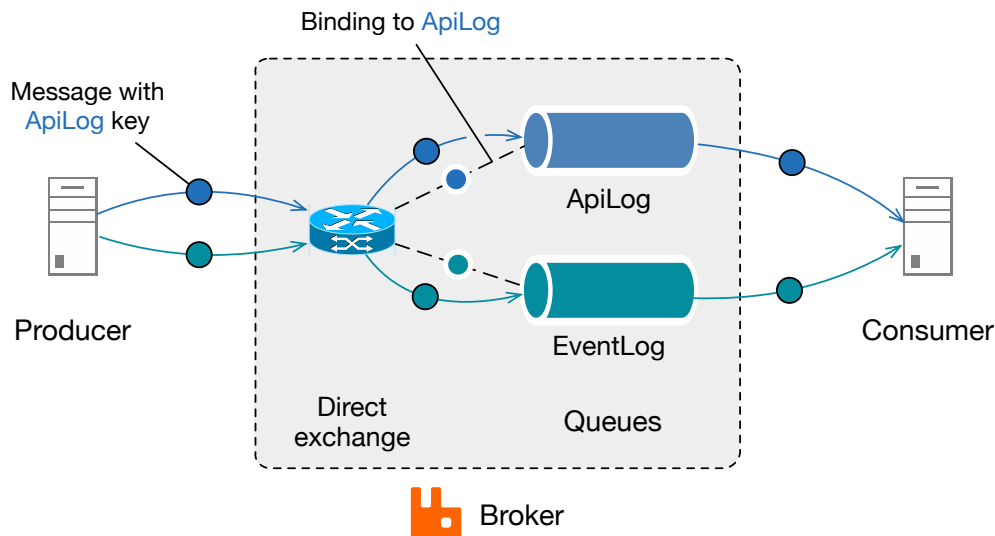


Figure 24 - RabbitMQ queue architecture

The producer will send messages to a direct exchange, which has two bindings whose routing key are for each one the name of one of the two queues, *ApiLog* or *EventLog*. Therefore, any message sent to the exchange with *ApiLog* as routing key will be delivered to the *ApiLog* queue. The same would apply for *EventLog*, while specifying any other routing keys would cause the message to be dropped.

Contrary to Kafka, RabbitMQ does not have a partition implementation. As a consequence, the consumer will run only two threads, one per queue. Each one will subscribe to one of the queues, which will cause the broker to push directly to the consumer data whenever a message comes in. The consumer uses the push API that RabbitMQ provides (see section 5.2.2 page 44).

5.5 Overall test architecture and load generation tool

The primary goal of the tests to follow is to assess the performance and reliability of the two message-oriented middleware presented so far, in comparison with the equivalent architecture without such messaging systems. It is expected that relieving the API from the write operations it has to do when an event is triggered should lower the global resource usage on the main Appgrid application.

To test Appgrid in adverse but yet realistic conditions, a high number of HTTP requests need to be generated toward the API server to simulate users. The test architecture is presented on Figure 25.

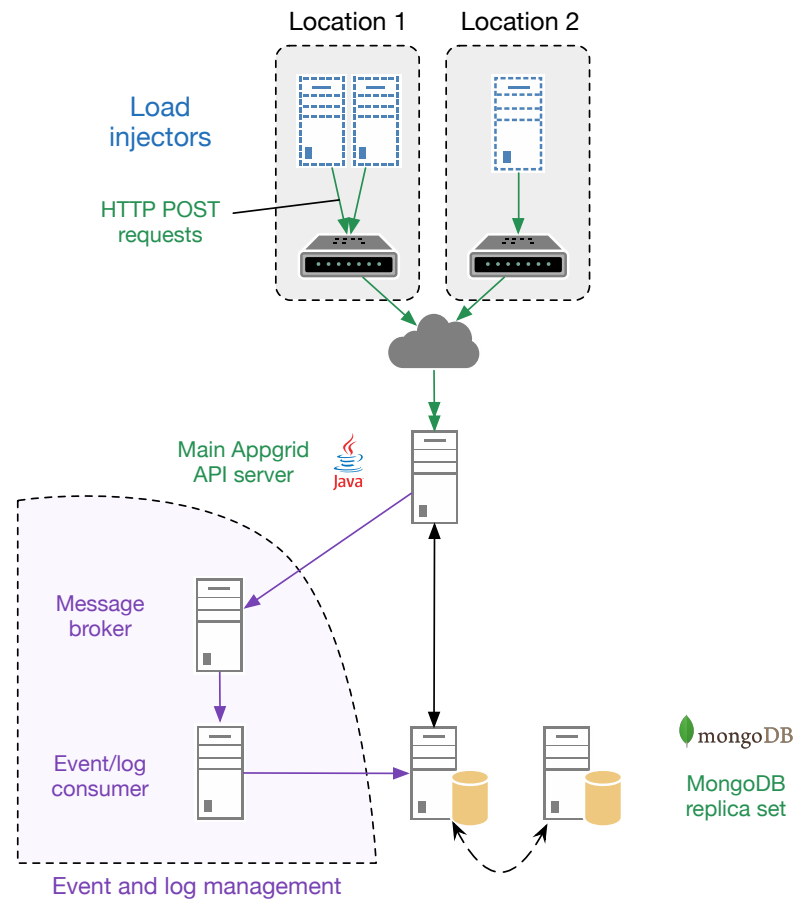


Figure 25 - Architecture used for load testing

There are a few differences with the architecture in production and the one proposed in section 3.5:

1. There is only one API server

The administration UI is not needed for the tests. It would not impact the results, and it was therefore not running at all.

2. The load balancer is not used

Since there is only one API server, the load balancer is not needed. Using only one server allows more load to be injected in a single Appgrid instance with the same load injection capacity. It also makes identifying possible bottlenecks easier.

3. Clients are load injectors

Three clients (computers) were used to generate load toward the API server. Two of them were located somewhere in Stockholm, using the same Internet connection, while the third one was also located in Stockholm but at a different place, to prevent any local network trouble and latency from interfering with the benchmark's results.

The Appgrid API server would run either the original version of Appgrid, without message bus support, or the updated one presented in section 5.2.1. The broker would be either Apache Kafka or RabbitMQ and the consumer would run the process described in section 5.2.2. The MongoDB replica set was an exact copy of the one that was in production at that time and hosts both the logging and the application (e.g. the metadata served by Appgrid, the active session keys) databases.

Clients here are load injectors. A software called *Tsung* was used to define a test scenario, federate the clients for them to play the created scenario and generate load on the server [63]. *Tsung* is an Erlang tool that makes distributed load testing easier. Probes and sensors can be installed on machines to monitor the activity of processes and statistics are generated at the end of the tests.

Other tools like *JMeter* and *httperf* from HP were considered, tested but eventually rejected due to either performance matters (*httperf* was performing very well when testing locally, but was in trouble with remote testing on Mac OS X) or complexity (extracting meaningful metrics from *JMeter* was a bit difficult) [64], [65].

5.6 Test scenario

The test scenario is quite simple: since there is a POST method in the API whose mission is to write an event to the database, it was obviously the easiest way to get things started and benchmark the messaging system. In the following, a transaction will refer to the scenario presented on Figure 26.

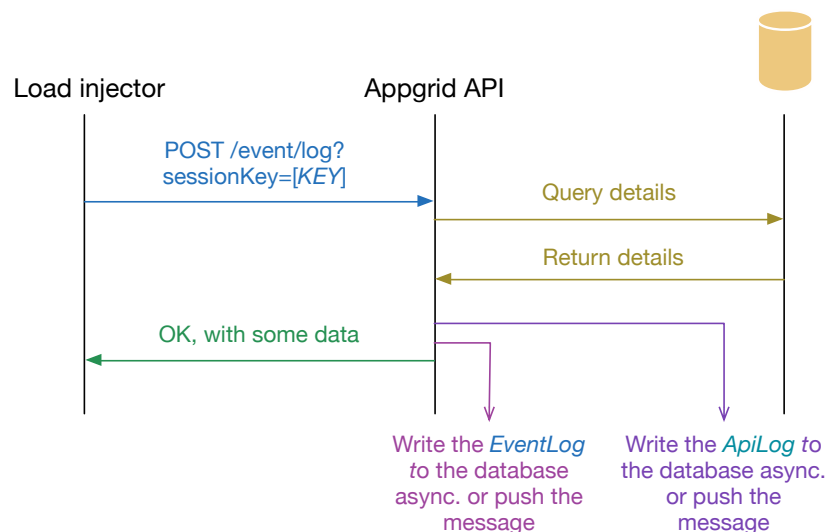


Figure 26 - Test transaction

A transaction begins with a TCP connection established toward the server, to perform a POST HTTP request to the URL `/event/log?sessionKey=[key]`, that is used by applications to push a notification to the server (e.g. when the application is terminated). The method called by this request needs to fetch data from the database about the session key to check whether it is authorized to perform this action (freshness) or not. Once it has it, it would create the *EventLog* object that needs to be inserted into the database. Note that the session key used is always the same, for all the transactions, which is likely not what would happen in reality, but appropriate enough to test the performance of message-oriented middleware.

What happens then differs depending on which Appgrid version is used: if the original Appgrid infrastructure (without messaging system) is running, the message is sent to a Spring asynchronous thread to be written to the database. If the new Appgrid infrastructure is in use, the API prepares and sends a message to the appropriate queue/topic.

However, even though the function performed by this method is to put an event in *EventLog*, it is still an API call. Therefore, calling this method triggers an event that needs to be stored in *ApiLog* as well. The way this is to be done is similar to the one described earlier for *EventLog* objects.

Eventually, the transaction finishes when the API returns a 200 HTTP code together with the queried information in the database (which includes among other information about the application linked to the given session key), and the TCP connection is terminated.

5.7 Load profile

To perform load testing, a load profile must be defined. In the studied case, the load profile was defined as a succession of steps, each one being set to last a certain time. Those steps correspond to a user rate, at which users are generated and connecting to the API. Each user is responsible for performing one transaction. The load profile that is expected to be generated by all the load injectors is shown on Figure 27.

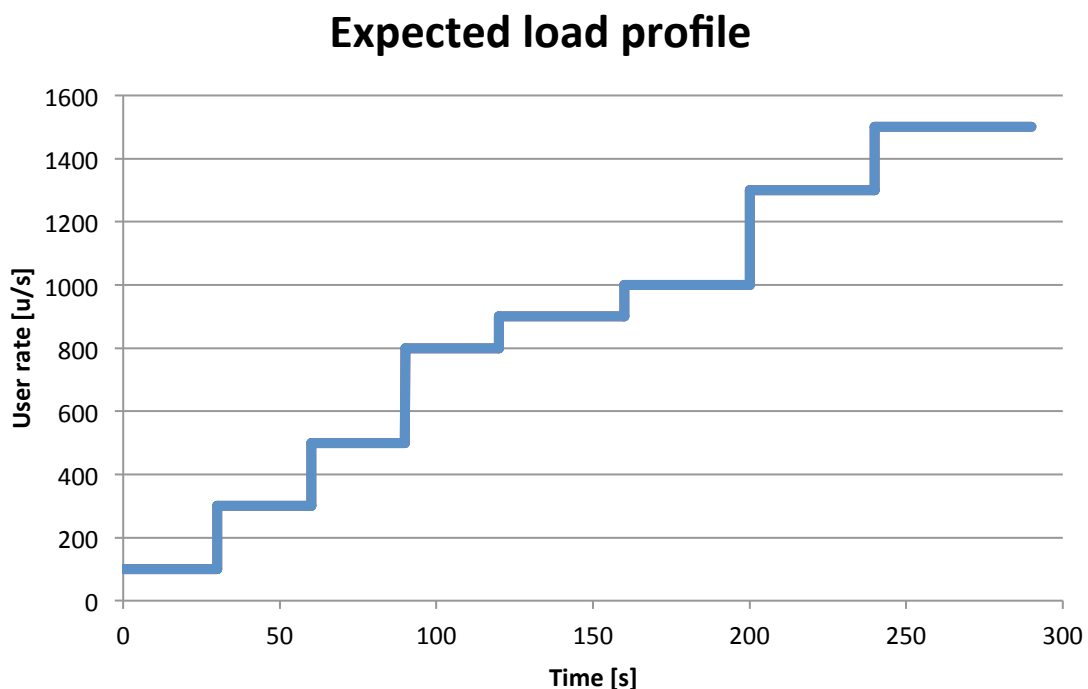


Figure 27 - User load that should be generated vs. time

The test scenario duration is fixed to 290 seconds. Most of the steps are 30-second long, while the ones with a larger user connection rates are longer to make sure that the server can handle such a load for a certain time.

The load profile is said “expected” as there is no guarantee that neither the load generation capacity will be enough, nor Appgrid will be able to handle such a large number of requests.

To compare with the historical numbers extracted from the statistics and surveillance tools that monitor Appgrid production servers, this load profile is really aggressive. The average number of requests per minute over a 3-month period (between mid-April and mid-July 2014) is of about 400, which means 7 requests per second. Over this same period, apart from an unusual peak at 2,400 requests per minute (40 requests per second) one day, the number of requests the API has to handle under normal conditions is way lower than even the first step of the proposed load profile. However, as Appgrid is expected to get more and more used, knowing the limits of a single instance is interesting to give system architects visibility and have a scalability plan.

5.8 Tested configurations

To compare the performance of the different brokers with different parameters, five configurations are defined and listed in Table 5.

Architecture	Original Appgrid	Kafka S	Kafka A	RabbitMQ NP	RabbitMQ P
Broker	None	Kafka	Kafka	RabbitMQ	RabbitMQ
Queues	None	EventLog, ApiLog	EventLog, ApiLog	EventLog, ApiLog	EventLog, ApiLog
Consumption acknowledgements	N/A	~ Implicit	~ Implicit	✓	✓
Production acknowledgements	N/A	✗	✗	✗	✗
Persistence	N/A	✓	✓	✗	✓
Producer push method	Async (Spring)	Sync	Async	Sync	Sync
Consumer message retrieval method	N/A	Pull	Pull	Push	Push
Message consumer	N/A	✓	✓	✓	✓

Table 5 - Test configurations' summary

All the tests with brokers involve the two queues mentioned in section 5.4, *EventLog* and *ApiLog*. The names of the architectures are simple: the difference between the two Kafka's configurations is the push method used by the producer to publish messages. *Kafka S* stands for *Kafka Synchronous* and *Kafka A* for *Kafka Asynchronous*. The same applies for RabbitMQ about persistence: *RabbitMQ NP* stands for *RabbitMQ Non Persistent*, while *RabbitMQ P* is for *RabbitMQ Persistent*.

6 Test results

The test architecture and configurations presented in section 5 have been loaded using *Tsung*, and this section presents the results of those tests and how they were obtained. Some specific details about the tests are listed for the sake of sustainability and verifiability. If the reader is interested in using *Tsung* to load his own application, he may find useful information in Appendix C — page 107.

6.1 Testing hardware and conditions

The *Tsung* load generators were two MacBook Pro (2006 and mid-2010) and one MacBook Air (2011). They were located in two distinct places, both equipped with 100 Mbps wired WAN access via optical fibre. The test server infrastructure was hosted on Amazon's EC2 instances, just as the production environment [66]. All the servers were on the same private network at Amazon, at the same geographical location. Table 6 presents the specifications of the Amazon's instances used for the test:

Instance	API server	Broker	Consumer	MongoDB instance
Amazon denomination	c3.large	c3.large	c3.large	m3.large
CPU	2x Intel Xeon E5-2680 v2 (2.8 GHz)	2x Intel Xeon E5-2680 v2 (2.8 GHz)	2x Intel Xeon E5-2680 v2 (2.8 GHz)	2x Intel Xeon E5-2670 v2 (2.5 GHz)
Memory	3.75 GiB	3.75 GiB	3.75 GiB	7.5 GiB
Storage	2x16 GiB SSD	2x16 GiB SSD	2x16 GiB SSD	1x32 GiB SSD

Table 6 - Hardware specifications of the server instances used for the tests

6.1.1 Java Virtual Machine and JIT compilation

The tested application is a Spring Java application. It is run on Java Hotspot Java Virtual Machine, and as such benefits from the optimizations that this JVM features [67]. One of these optimizations deals with Just-In-Time (JIT) compilation. Java bytecode is converted to machine code on the fly, when the program is run. The first time this conversion occurs, the main objective of the compiler is to produce code as quickly as possible, thus skipping advanced optimizations that it could make. The code is working but could probably be better if the compiler spent more time analysing and optimizing it.

This is where the “adaptive optimization technology” comes onto the scene: this feature, part of the Hotspot JVM, passively profiles the application and detects which parts of the code, which methods are called often and will recompile those code sections to produce a more optimized code. Typically, software runs only few parts of the code regularly, while most of the program would almost never be invoked. With *adaptive optimization technology*, the compiler is able to produce applications that start quickly, and that get better with time.

This “warm-up” time must be taken into consideration while testing. Note that the compiler detects and reruns optimization after profiling for some machine cycles, and not user time [67, Ch. 3]. Therefore, the tests included before launching the real measurements a “warm-up” test, which was simply the same as the test to be executed, while following the activity of the JIT compiler. Once every metric was stable enough, the test was interrupted, and the real one was launched, after a short pause.

6.1.2 Test reliability

The benchmark needs to be reliable. To fulfil this requirement, the tests were run multiple times (at least three times) with and without restarting the Java Virtual Machine, to make sure that the results were not fluctuating in large proportions. The most significant test was eventually kept, and used as a basis for further analysis.

6.1.3 Producer, broker and consumer settings

Here follow the parameters used in the brokers and the consumers:

- Kafka producer, maximum number of messages in the async. queue¹: 10,000 messages.
- Kafka producer, message batch size capacity: 200 messages.
- Kafka producer, message batch timeout: 5 seconds.
- Kafka mode of operation on full local queue (async. mode): block and wait for the queue to offer free space.
- RabbitMQ prefetch count (per queue): 500 messages.
- Consumer *DBObject* queue length (per thread): 500 objects.
- Consumer flush queue timeout (per thread): 1 second.

6.2 Followed metrics and instrumentation

To get the widest picture as possible of the overall architecture, multiple tools were used to follow a large number of metrics that would be of interest when comparing the performance of the different configurations. Here follows the list of the different metrics, with the tool used to monitor and record them, per instance:

- **Appgrid API server**
 - **Tools:**
JVM monitored via JMX [68]. VisualVM with Tracer and MBeans Browser plugins was used to have a convenient way to monitor different metrics on the Java instance, and export them to CSV afterwards [69].

¹ There is a subtle difference between the queue size and the batch capacity: if the producer generates more messages than what the Kafka library (in asynchronous mode) can send, the process may pile messages up to the queue size. The batch capacity is the maximum number of messages that are sent in one operation to the broker. See section 4.2.2 page 30 for more information.

- **Metrics:**
 - CPU usage of the Java instance.
 - Memory usage of the Java instance.
 - Garbage Collector activity of the Java instance.
 - Number of threads alive, maintained by the Java instance.
 - Number of file descriptors opened by the Java instance.
 - Java compiler activity.
- **Kafka only:**
 - Cumulated number of messages received per topic (MBean).
 - Cumulated number of messages received in all topics (MBean).
- **Kafka broker**
 - **Tools:**

As mentioned in section 4.2.3, Kafka relies on MBeans to expose its metrics. VisualVM with Tracer and MBeans Browser plugins was used to fetch those metrics but from the producer side, as only one JMX console could be used to reach a JVM at a time (metrics about Kafka were thus read in the MBeans exposed by Kafka's library in the Appgrid API instance). *sar* (System Activity Report Unix command) was used to get system metrics [70].
 - **Metrics:**
 - CPU usage on the machine (including I/O wait).
 - Memory usage on the machine.
- **RabbitMQ broker**
 - **Tools:**

RabbitMQ has a fully featured web interface and API that allows metric extraction. The *rabbitmq-management* plugin was used, together with *sar* to get system metrics [57], [70].
 - **Metrics:**
 - CPU usage on the machine (including I/O wait).
 - Memory usage on the machine.
 - Cumulated number of messages received per queue.
 - Cumulated number of messages received in all queues.
 - Number of queued messages at a given time per queue.
 - Number of queued messages at a given time in all queues.
 - Publishing rate per queue (i.e. rate at which messages were received from the producer).
 - Publishing rate in all queues.
 - Delivery rate per queue (i.e. rate at which messages were sent to consumers).
 - Delivery rate for all queues.
 - Acknowledgement rate per queue (i.e. rate at which acknowledgement of consumption arrived from consumers).
 - Acknowledgement rate for all queues.

- **Consumer (RabbitMQ or Kafka)**

- **Tools:**

- Though MBeans are exposed by Kafka in the client library just as they are in the producer library, the limitation of one JMX console available during the tests did not allow reading them. Instead, verbose logging was enabled on the consumer process, which printed a line each time a queue was flushed to the database, together with a timestamp and the name of the queue. Parsing those logs could give some data. *sar* was used to get system metrics [70].

- **Metrics:**

- CPU usage on the machine (including I/O wait).
 - Memory usage on the machine.
 - Cumulated number of documents inserted into the database per collection.

Beyond system metrics, *Tsung* gives several indicators regarding the test itself and the HTTP operations. Here are the metrics that were followed (available with a 10-second interval):

- Number of users connected.
- Number of users that completed their transaction.
- Mean transaction time.
- Mean transaction rate.

6.3 Data analytics methods

To compare performance and metrics between the different test configurations, figures must be homogenous. Performance indicators like CPU usage are difficult to compare from one test to another, as traditional statistics tools like min/max and average are of little interest here.

Nevertheless, figures must be compared to have a quantitative estimation of the performance of the various test configurations. RabbitMQ configurations were the easiest to analyse: the management interface gives all the needed metrics at whichever specified time resolution. Message rates were directly captured from the interface and compared against each configuration.

For Apache Kafka, the gathered metrics are of very little interest and are highly inhomogeneous. For instance, if on average, the interval between each value gathered from the MBeans described in section 6.2 is of 5 seconds, this far from being the usual case. Sometimes, values are separated by 0.2, 18 or 24 seconds, to give some examples. Moreover, no message rates were available, but just the cumulated message count. Rates were therefore inferred from the count, by either calculating a variation rate between two instantaneous values between each step (numerical derivation), or by making a linear approximation when the rate was almost constant (used at the consumer side).

Some values have been removed from the tests with Kafka, as the JMX console gave very unrealistic results when the garbage collector was acting. When such a garbage collection event happened, the interval on which the rate was calculated was extended to cover the whole garbage collection time plus the value before and the value that followed. In this way, the value was *averaged* on quite a long time, which preserved the general shape of the curve.

CPU comparison was done by numerically integrating (using the trapezoid rule [71, Sec. 9.2]) the CPU usage curve over the course of the test. By itself, the figure obtained after this operation is absolutely meaningless, being the area between the x-axis and the CPU usage curve. However, this value can be used as a reference for comparison between the same CPU curves, captured during the same time interval, and integrated in the same way. The relative difference between these values would then give a quantitative idea of the improvement (or issues) one configuration might have over another.

The original Appgrid API was defined as the reference for CPU usage comparison of the core API application. The value the numerical integration of the CPU curve was said to be an index set to 100, while the other curves (for other test configurations) were indexed against the original Appgrid API CPU usage. It means that if configuration A has a corrected CPU usage index of 80, then configuration A consumed over the course of the test 20% less CPU than the original Appgrid API. If the index was 150, then it required 50% more CPU over the test course. Note that all CPU usage percentage metrics are defined as $CPU\ Usage = 1 - Idle\ CPU$.

The same rule for calculation was applied for consumer and broker CPU consumption comparisons. However, the reference could not be the original Appgrid API, as there is no consumer and broker in these cases. Arbitrarily, the reference has been set to be Kafka using synchronous push method on the producer side. It therefore has a corrected CPU index usage of 100, while the other configurations are compared against it.

Metrics' sources were quite different from each other. Python scripts were used to parse RabbitMQ JSON structure, capture the interesting metrics, compute new ones and convert them to CSV files (refer to Appendix B — page 104). *Tsung* results were used as generated. VisualVM Tracer plugin allows easy CSV export of the monitored metrics. All the data was gathered into a single Excel workbook per tested configuration. Statistical analysis was made using Microsoft Excel.

6.4 Appgrid performance results

Table 7 presents the aggregated results from the tests about the Appgrid API performance metrics.

Architecture	Original Appgrid	Kafka S	Kafka A	Rabbit MQ NP	Rabbit MQ P
Number of users generated	168,488	179,278	175,358	176,162	175,656
Test duration [s]	290	290	290	290	290
Mean transaction rate [req/s]	581	618	604	607	605
Maximum 10-sec mean transaction rate [req/s]	788	854	886	874	850
Mean transaction time [ms]	2,923	1,490	2,186	908	1,394
Maximum 10-sec mean transaction time [ms]	21,388	8,129	17,723	5,145	13,481
API CPU usage index ¹	100	92.1	82.5	79.6	85.8
Number of ParNew GC ² runs	8	1	1	0	1

Table 7 - Synthetized test results about the Appgrid API

Colours have been added to make it easier to identify the best (in green) and worst (in red) configuration for each metric. The test settings and configurations can be found in section 5.8 page 55.

The first conclusion that can be drawn is that messaging systems always make the API performing better, while consuming less CPU. The original Appgrid architecture is outperformed in all the followed metrics, sometimes to great extents.

One of the primary explanations to this situation is that garbage collection occurs fewer times when messaging systems are used. The original Appgrid API performed 8 ParNew GCs during the test. During about 120 seconds of the test, the JVM was busy cleaning the memory, which made the application really slow and unresponsive. It resulted in a large number of timeouts and connection errors (about 4,000) and in an important increase of the maximum 10-sec mean transaction time. The 10-sec means are means calculated only on values from a 10-sec interval, without taking into account the values before or the ones that follow. In the case of the transaction rate, it then reveals the best performance of the API (usually under low user load), while the mean transaction time is likely maximized when the load is high (i.e. in adverse conditions).

The best performer according to tests is RabbitMQ when persistence is disabled. Under the same load, no stop-the-world garbage collection was initiated, which has greatly decreased the overall CPU usage of the API, by more than 20%. The mean transaction time was below 1 second, thus 3 times lower than in the original API. The maximum 10-sec mean transaction time was also 4 times lower than with the original architecture.

¹ Reference: Original Appgrid = 100.

² See glossary page 88.

Persistence, while being something that is taken care at the broker side, impacted the performance on the producer side as well with RabbitMQ. Indeed, as shown in Table 9 below, the broker becomes far more stressed with persistence enabled and therefore takes more time to answer, while the pressure on the API remains the same. As a consequence, the API generally responds slowly, and consumes more resources. The reader should remember that messages are sent synchronously with RabbitMQ, and that Appgrid is therefore dependant on the broker's ability to accept message quickly to be able to answer HTTP queries quickly.

Kafka on the other hand always stores messages. Compared with RabbitMQ with persistence enabled, they present similar performance. The main difference between the two configurations being the method used to push messages to the broker, the asynchronous version was excepted to present better performance, since it would batch messages before sending them. But if more requests can be sent toward the API, they take on average almost twice as much time to get processed. This may be explained by the mode of operation on full queue specified in section 6.1.3: if more messages are produced than what the Kafka library is able to send to the broker, the thread which wants to push a new message get stuck until the message can eventually get pushed to the local queue. However, it is very unlikely that 10,000 messages got stuck in the local queue. Another reason may explain this curious latency, but has not been found yet.

Comparing the two message-oriented middleware on those metrics does not designate a real “winner”: they both perform well, with a special plus for RabbitMQ without persistence, which is the best in almost all the tests.

Apart from these brokers' considerations, it is worth noting that testing the original API hit very quickly some kernel limitations, in terms of number of file opened by the system, and the same limit by process. The *java.io.IOException: Too many open files* shown up as soon as the number of simultaneous connections became larger than 200 (sockets are handled as file descriptors in Unix systems), and therefore required the kernel limitations to be raised to handle more load (via *ulimit -n*).

6.5 Broker specific results

Table 8 presents some figures about message streams.

Broker specific results	Original Appgrid	Kafka S	Kafka A	Rabbit MQ NP	Rabbit MQ P
Maximum global publishing rate [mess/s]	N/A	2,591	3,177	1,799	1,477
Maximum global database insertion rate [mess/s]	N/A	1,241	1,270	1,307	1,187
Maximum global delivery to consumer rate [mess/s]	N/A	?	?	1,337	1,160

Table 8 – Synthetized results about message rates

The global pushing rate is the combined rate at which the API sends messages to the broker. For Kafka, this value is measured on the producer process, while for RabbitMQ it is measured on the broker side. Kafka values are almost twice as important as the values one can obtain using RabbitMQ with persistence enabled. However, the figures for Kafka should be taken with caution, as they were only available via MBeans, which did not give values on a regular basis. It appears clearly on Figure 28 that Kafka's metrics were clearly not stable, while RabbitMQ presents a steadier curve, in line with what is expected.

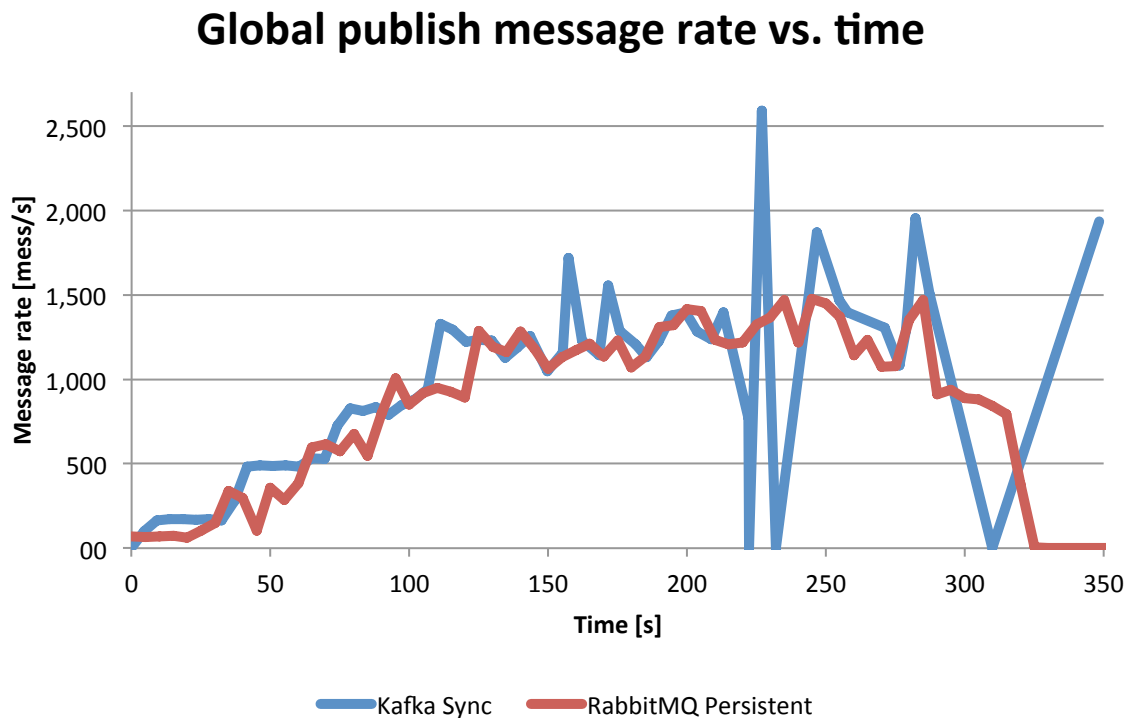


Figure 28 - Comparison of the global message push rate between Kafka in synchronous mode and RabbitMQ with persistence

One can wonder why there are messages that are still sent after the end of the test (which is meant to finish at 290 seconds). It is just because the test is brutally interrupted at 290 seconds, but since the average request time at the end of the test is usually high (in the magnitude of 10 seconds), the connections that were established when the test was interrupted were still in process on the server side, which take some time to complete. Therefore, the number of messages sent decreases slowly and then reaches zero after a short while. The end peak with Kafka is another artefact of the JMX value processing.

The second metric presented in Table 8 is related to database insertion. It is the result of a log parse on the consumer process that prints a line each time the *DBObject* queue is flushed to the MongoDB instance. The variation are as expected quite limited, since the consuming process implementation is the same, no matter the broker. The only change is the client library used by the consumer to fetch or receive messages, and it does not seem to impact performance. Note the difference between the insertion rate and the publish rate that illustrates the “shock absorbing” role of the message broker.

On Figure 29, that plots the total amount of message in all queues for RabbitMQ¹, this buffer effect is all the more visible. While the system was tested on relatively low load (between 0 and 110 seconds), the consumer was able to follow the stream and almost no messages were queued. After 110 seconds and until the end of the test, more messages were received, exceeding the consumer's capacity, and therefore many got queued. At the end of the test, the queue is emptied by the consumer at full and constant rate (the number of queued messages decreases linearly).

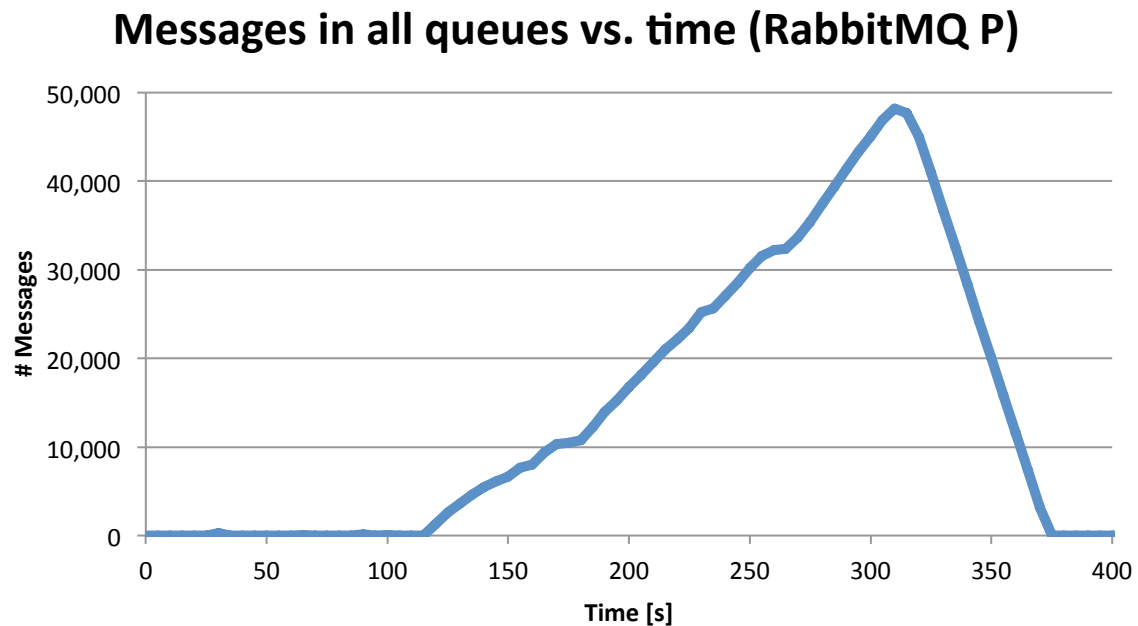


Figure 29 - Total number of messages (in queue + not acknowledged by consumer) vs. time, for RabbitMQ with persistence

The last metric, only measurable for RabbitMQ, is the rate at which the broker delivers messages to the consumer. What is interesting with this metric is the comparison with the database insertion rate: if they are approximately equal, it means that the consumer is able to process data at the same rate at which it receives it, meaning that data stays in transit at the consumer side only for a short time. This is the case here, but the result is a bit artificial, as the rate at which the broker sends message to the consumer is throttled by a rather conservative prefetch count (see section 6.1.3 page 57).

¹ Remember that RabbitMQ does forget about a message once it has been consumed. Therefore, a message is in queue only if no consumer has acknowledged its consumption (or if it has not been delivered to a consumer at all). See section 4.4 page 39.

6.6 Broker and consumer performance

Table 9 presents the CPU usage indexes of the brokers and consumers:

Broker and consumer performance	Original Appgrid	Kafka S	Kafka A	Rabbit MQ NP	Rabbit MQ P
Consumer CPU usage index ¹	N/A	100	97.1	86.7	93.7
Broker CPU usage index ²	N/A	100	49.9	202.1	1,038.2

Table 9 - Consumer and broker performance comparison

One very important point when installing a new system in a production architecture, especially when its primary mission is to ease scalability, is to be sure that it will not create more problems than those it is meant to solve. Therefore, the behaviour of both the broker and the consumer were followed with interest and compared.

Apache Kafka presents top-rank performance values, both for the consumer process and the broker. Specifically, the broker is very efficient, and the asynchronous version even halves the overall CPU consumption of the broker compared to its synchronous version, as shown on Figure 30. It is also worth noting that almost no I/O wait CPU time is experienced while running Kafka, thanks to its *pagecache* method of writing data to the disk (see section 4.1 page 26).

Kafka broker CPU consumption vs. time

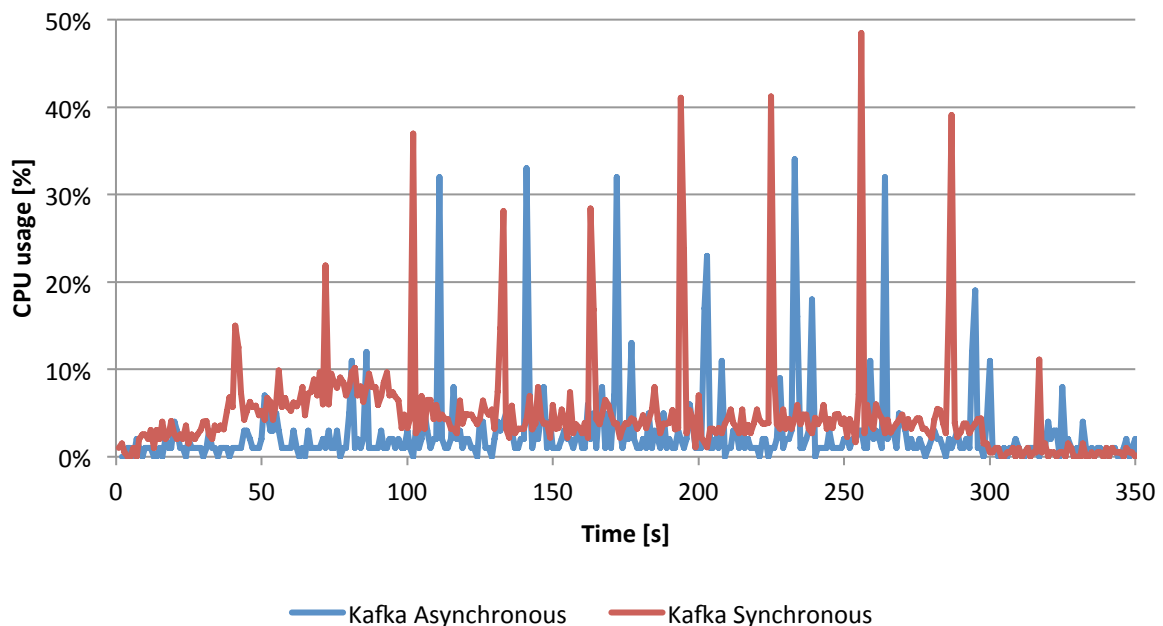


Figure 30 - Broker CPU usage (for Kafka) vs. time

¹ Reference: Kafka Sync = 100.

² Reference: Kafka Sync = 100.

On the other hand, RabbitMQ's broker consumes more CPU. Twice as much as Kafka in non-persistent mode, and as much as 10 times more when persistence is enabled. Remember that RabbitMQ pages out to the disk only messages that cannot be delivered and acknowledged right away by a consumer, or messages that exceed the available space in memory (see section 4.3.2 page 36). Since the space available in memory was way larger than the total size of all the delivered messages, messages were written down to the disk because they exceeded the consumer's absorption capacity (as shown on Figure 29 page 64). Figure 31 compares the two RabbitMQ configurations in terms of CPU consumption. Pay attention when comparing this graph with the one for Kafka to the fact that the y-scales are different.

RabbitMQ broker CPU consumption vs. time

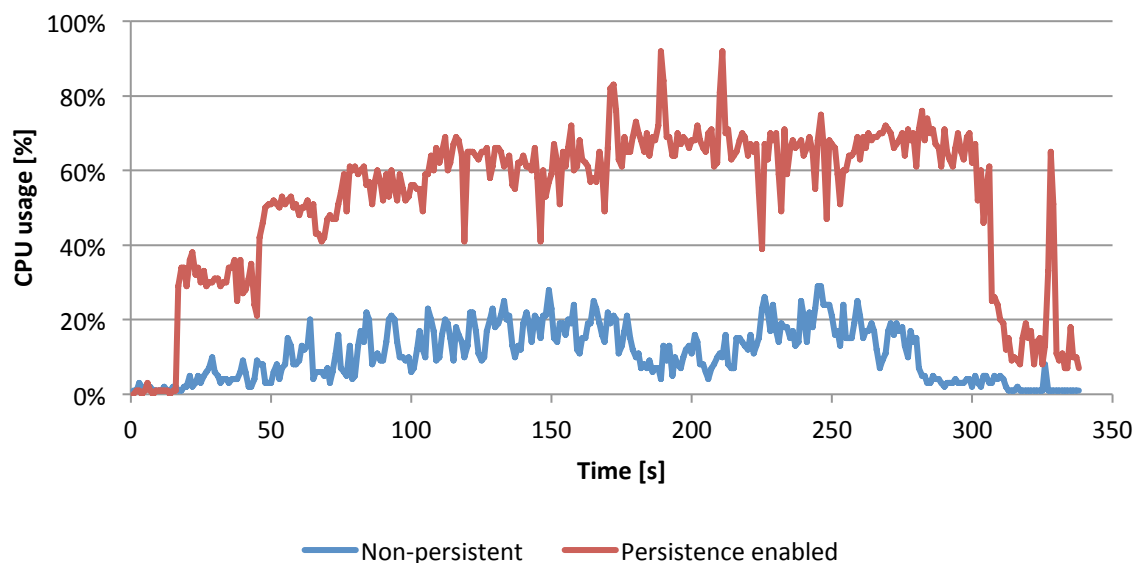


Figure 31 - Broker CPU usage (for RabbitMQ) vs. time

Remember that the CPU usage metrics plotted above are defined as $CPU\ Usage = 1 - Idle\ CPU$ (see section 6.3 page 59). Therefore, it comprises the CPU I/O wait time, which is very high when persistence is enabled. The arithmetical mean of the I/O wait CPU time percentage is as high as 26,2% over the course of the test when persistence is enabled, while being below 0,2% without. Therefore, one quarter of the CPU time is wasted waiting for the disk. Persistence really hurts the overall performance of the broker instance, and enabling it should be avoided whenever possible. Having a machine with higher I/O capacities would help reduce the resource consumption on the broker side.

The consumer process was also monitored, and presents similar performance no matter the broker it has to communicate with. This is largely explained by the fact that the CPU is highly solicited throughout the course of the test in all the configurations, as shown by Figure 32.

Consumer CPU usage (RabbitMQ NP)

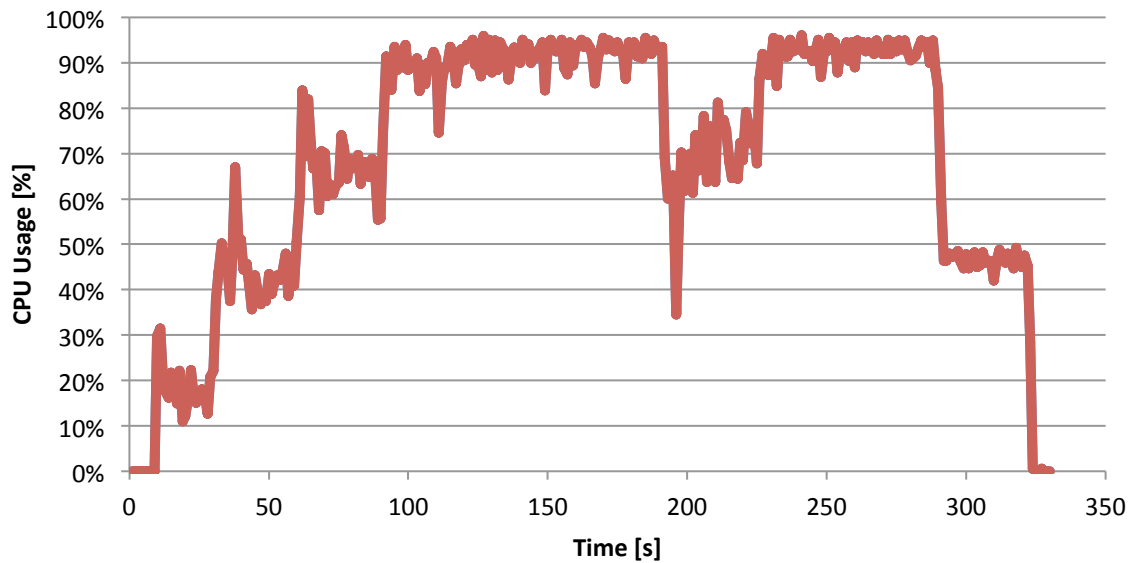


Figure 32 - CPU usage by consumer process (with RabbitMQ non-persistent)

According to the figures in Table 9, the configuration corresponding to the CPU consumption presented above is the one that uses the less resources. Though, it appears that the CPU is fully busy during most parts of the tests, and therefore highlights an implementation bottleneck in the consumer's code. Tweaking queue flushing parameters (regarding bulk inserting) may lead to better performance. Database performance limits may have also been hit. And since the consuming process does not monitor nor react to the database's load, it may be overwhelming the database, which therefore takes longer times to answer.

Nevertheless, it is worth noting that a version of the consumer without bulk inserting (and with only one collection to be filled, while the bulk insertion figures comes from the two-queue scenario studied so far) was tested. Figure 33 shows the comparison.

One-by-one vs. bulk insertion (Higher is better)

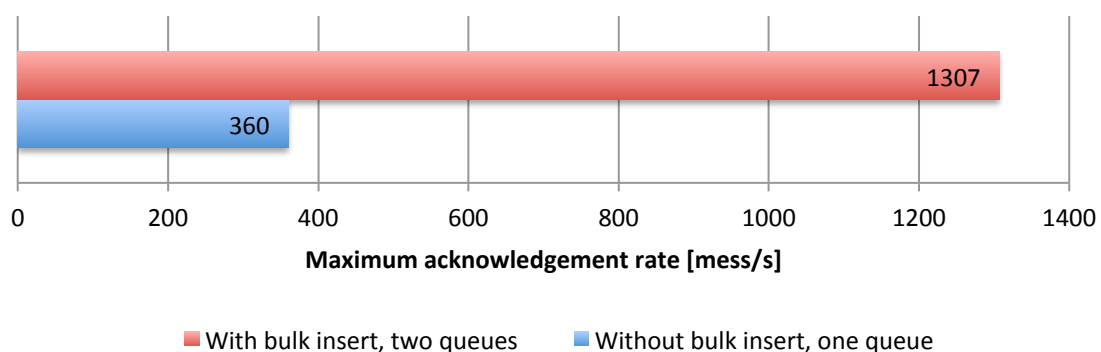


Figure 33 - Benefits of bulk insertions into MongoDB vs. single insertion (using RabbitMQ without persistence)

Despite the fact that the consumer was more loaded when it had to do bulk insertions, it could absorb 4x more messages than the single-queue consumer process that was doing one-by-one insertions used at the beginning. One may notice that the maximum acknowledgement rate in the single queue case was way lower than the maximum publishing rate reached (about 700 messages per second in this test, which is not presented in the report as being irrelevant for the comparison). The consumer therefore had to run for almost 300 additional seconds after the end of the test to empty the queue completely. While illustrating the ability of message-oriented middleware to absorb traffic bursts, it also revealed how inefficient the consuming process was, and how important it is to design a consumer process that takes completely advantage of the architecture.

6.7 Test conclusions

These tests in a real environment show how interesting it is to implement a message-oriented middleware in a typical, monolithic architecture with self-hosted asynchronous threads: performance improvements, efficiency, flexibility, load balancing, burst management... In all the tested configurations, Apache Kafka and RabbitMQ relieved the main API from asynchronous processes that insert data into a MongoDB instance, yielding overall performance improvements. RabbitMQ without data persistence won the API performance race, by being able to reduce by 20% the CPU power consumed by the API compared to the original one, while managing to keep the average transaction time below a second when calculated over the test course.

However, the performance of the broker instance revealed something different: Apache Kafka is in fact more CPU-conscious and consumes in all situation at least twice as less CPU as its RabbitMQ counterpart, while writing everything to the disk. RabbitMQ, with persistence enabled, revealed awful performance, mainly due to poor I/O capacities on the server hosting the broker, but also highlighting the inefficiency of the persistence method that RabbitMQ relies on.

Consumer performance was not affected by the broker used, but rather by the way messages were processed. The broker cannot do anything against other performance bottlenecks in the architecture (e.g. database capacity). However, by taking advantage of the fact that messages can be processed in batches at the consumer side, powerful bulk features can leverage great performance improvements, and make the whole architecture even more powerful.

These tests did not study the impact of the message size on the performance of all the tested systems, as the size was fixed by the application logic and could not be possibly changed. However, it has been shown in other studies that it has an importance, and that this criterion should be considered when selecting such a system [35].

7 Leveraging the advantages of Message-Oriented Middleware for data analytics

This section presents how implementing a message-oriented middleware together with a data warehouse system in Appgrid enabled the overall data aggregation and analytics process to be more reliable, flexible and especially way faster than it was. It focuses on RabbitMQ and the hosted data warehouse system provided by Amazon, Redshift, to demonstrate the benefits of using messaging to achieve nearly real-time data aggregation.

7.1 Current aggregation process in Appgrid

As presented in section 3.1 page 19, the original Appgrid architecture relied on a MongoDB database to store all the information regarding the events managed by Appgrid. Appgrid itself was relying on the Aggregation Pipeline of MongoDB and Map-reduce jobs to aggregate the data (i.e. process the raw events to produce meaningful data) [72], [73]. An example of what “aggregation” means is given on Figure 34:

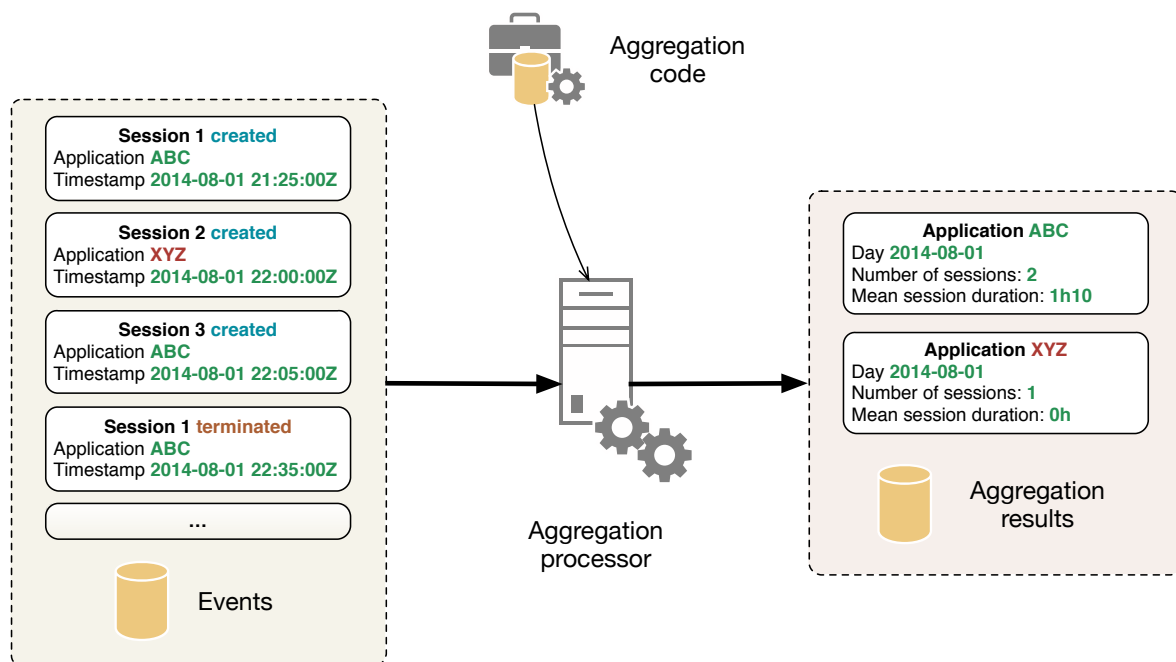


Figure 34 - General aggregation process

In the current architecture, the *events* part on the left-hand side of Figure 34 consists of all the raw event logs stored as documents in few collections of the MongoDB replica set. Specific and rather complex aggregation codes are sent to MongoDB to be run by the aggregation processor (which is currently the MongoDB replica set as well). This processor takes the input logs, parses them and outputs the statistics that the aggregation code was set to calculate. For instance, event logs can be parsed to produce one document or *row* per application that produced logs, summing the number of sessions created and telling their average duration, on a daily basis.

One of the main advantages for customers to use Appgrid is its ability to provide these figures and statistics about application usage very easily, without having each developer to build his own system. The more metrics are followed, the more customers are likely to find one that fits their needs. Therefore, Appgrid aggregates the events it manages in multiple ways, on various time periods (daily, weekly, monthly basis), on various grouping attributes (per application, per platform, per country, and a combination of those) and with various needs (counting records, summing counters, averaging durations, etc.). Computing these statistics is highly time and resource consuming.

To avoid an enormous and unnecessary load on the database due to queries made by Appgrid users on the logs, this aggregation process is batched, run several times a day and its output is saved into multiple MongoDB collections. The analytics front-end, served to the end user, is actually fetching already-crushed data from these collections, instead of running each time the process described in Figure 34. These collections are therefore represented by the *Aggregation results* part of the figure, on its right-hand side.

7.1.1 Problems with this solution

While being totally functional, this scheme has a number of disadvantages that prevents the system from being really scalable:

- **Aggregation computations are run on "online" and shared machines**
The queries to produce the aggregated data sets are sent to the MongoDB replica set that is actually storing the data. It means that the machines that are used and contacted by the Appgrid API are the same as those which, when the aggregation process is started, will crunch the logs. Since the aggregation process might take a while (20 minutes at least, but up to 2 hours in production), it is likely that the overall performance of Appgrid itself will suffer from a process that should be running in background and not impact the other tasks. No isolation is provided.
- **Complex and database-dependant aggregation code**
Some queries that must be run to produce meaningful data out of the raw logs can be quite twisted and difficult to come up with. While such processes may be difficult to describe with plain English words, it may be all the more difficult to translate it to programming code that the aggregation processor can understand. SQL, or Structured Query Language, is a widely used and standardized language in which those queries can be written [74]. However, as the data is stored in MongoDB instances (which is by definition a so-called *no-SQL* database system), this language cannot be used to define the queries, and should instead be written as MongoDB Map-reduce jobs, described in a combination of *JSON* and JavaScript files. This requires one to learn another language, which is not the most popular one to perform this kind of operations, and probably not the easiest to use when comparing equivalent queries written in SQL. The feature sets of the two alternatives are different, and one may be better in some cases than the others, but for the operations that matter here, writing MongoDB Map-reduce jobs can become quite difficult to read (one may find an interesting example of an SQL/MongoDB comparison in the references [75]). Such complexity in writing the queries also makes them even more difficult to debug.

- **Multi-step calculations**

Map-reduce jobs on MongoDB do not allow one to take advantage of some niceties that various relational database systems feature like Common Table Expression, or CTE [76]. This specific functionality (that is recognizable in queries as it is introduced by the keyword *WITH*) makes among other things possible for a query to use as its input the result of another one, whose results are computed when needed and destroyed when the final output is produced. CTE can help avoiding writing the results of intermediate query results to disk when it is only needed as input for another job to compute the final set of data.

For some large or complex calculations, MongoDB Map-reduce jobs require the use of intermediate collections, in which the output of intermediate queries are outputted and used by other Map-reduce jobs as input, to be eventually aggregated as wanted. It consumes disk space and may be fairly inefficient.

- **No easy “quick-and-dirty” testing**

Writing map-reduce jobs is really not as easy as writing SQL queries. Data scientists may be interested, in their quest of new metrics to be proposed to the customer, to make queries on the real data set using a simple query language they already master. Launching map-reduce jobs and interpreting the results is a time-consuming task, and prevents on the long-run creativity: one would stick to the metrics already computed to avoid the hassle of having to write and test new things. Doing the same in SQL is a matter of seconds with any kind of decent SQL console, and makes this “creativity” exercise opened to more people, even those with limited technical knowledge.

7.2 Proposed solution: split the work

It is obvious that MongoDB may not be the perfect tool to aggregate data on a large scale when complex data manipulations have to be done on the raw documents. The whole point of the thesis was to investigate how introducing a message-oriented middleware in the architecture could be beneficial for an application, specifically with data analytics matters. The idea is as depicted on Figure 35: split the tasks between specialized instances.

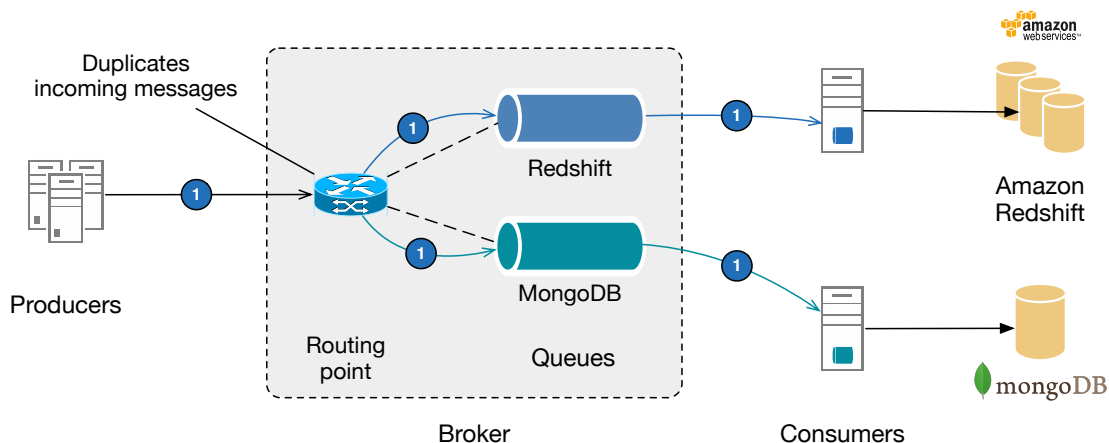


Figure 35 - Message-oriented middleware with Amazon Redshift

The process is simple: producers (i.e. the Appgrid API servers) send events to the broker only once. Those events are duplicated by the broker and stored in two different queues, one emptied by the already-studied consumer process that fills the MongoDB database (see section 5.2.2 page 44), and the other one by a new consumer process that will take the raw logs and store them into Amazon Redshift. As described in section 2.3, having two consumer processes and two queues isolates completely the storage of raw logs for display in MongoDB, and their storage for aggregation in Amazon Redshift. One process can be faster than the other, or even crash, without any impact on the other.

7.2.1 Amazon Redshift

Amazon Redshift is a data warehouse service that relies on a *lazy*¹ relational database scheme to store data in the most efficient way possible, and to make data manipulation both easy and fast [77]. Easy because the whole system is based on SQL: the database scheme is created and managed via typical *CREATE TABLE* commands, and data can be inserted using traditional SQL command sets. Amazon Redshift is derived from PostgreSQL and therefore inherits parts (some functions or features are not available, like timestamp with time zones and *DISTINCT ON* option in *SELECT* queries [78]) of its feature set, like its support for CTE. It also supports connections from typical SQL client and JDBC drivers, which facilitates a lot the interconnection of various tools to the system.

Amazon Redshift is also blazingly fast when the data it contains is queried, even if the data sets it needs to deal with are very large. Thanks to an intelligent load balancing work done behind the scene across multiple computing nodes (architecture known as Massively Parallel Processor architecture, or MPP) and its columnar storage system, the system is able to return query results within seconds, while other systems can require minutes or hours to do the same job [79].

Many small and big companies report their success stories and how happy they are with the services provided by Amazon Redshift [80], [81]. Some benchmarks compare this tool against others offering the same kind of aggregation facilities, like Hive or Shark, and reveal how efficient and polyvalent Amazon Redshift is [82]. The point of this section was not to compare the performance of different data warehouse and aggregation systems but rather to highlight how powerful and handy they were combined to a message-oriented middleware, and how efficient they were compared to the previous aggregation solution.

Amazon Redshift was eventually chosen for testing as it was SQL-driven, easily reachable from third-party tools (thanks to its JDBC compliance), well documented and also cheap: there is even a completely free trial version (as of August 2014) that allows one to run a cluster freely during two months. Google presented its own cloud-based aggregation service in 2010, Google BigQuery, but it was not possible to test it freely and it has a per-query cost that is a bit difficult to determine [83]–[85].

¹ Data are stored in tables that have a fixed number of columns and each has a type, but foreign keys and integrity constraints, though they can be defined, are not enforced by the system itself.

Table 10 presents the technical specifications of the test cluster, and an estimation of its cost (this section was valid as of August 2014). The tests were carried out using a trial version of Amazon Redshift, which was fully functional and free.

Cluster type	Single node
Node type	dw2.large
Storage	160 GB (SSD)
Cost (pay-as-you-go)	\$0.25/h, \$180/month, \$1790/year
Cost (reserved instances) for 1 year¹	\$1407 with 1-year commitment, \$880 with 3-year commitment
Extra costs²	\$10/month

Table 10 - Amazon Redshift specifications and cost

7.2.2 Database architecture

The MongoDB part of the architecture is not changed at all: the very same consumer process as the one presented in section 5.2.2 is being used in this section, and the same replica set is used as well, without any changes on the logs that are being stored.

The database in Redshift is however slightly different from the structure of the documents stored in MongoDB. Not all the fields contained in the logs are stored in the database: events containing application logs also carry a message, which is a string sent by the application developer. Appgrid will not provide any statistics on the value of this message field, and it may be quite large. Therefore, those kinds of fields were simply not inserted into Redshift. For information, the table model used in Amazon Redshift for the tests is presented on Figure 36.

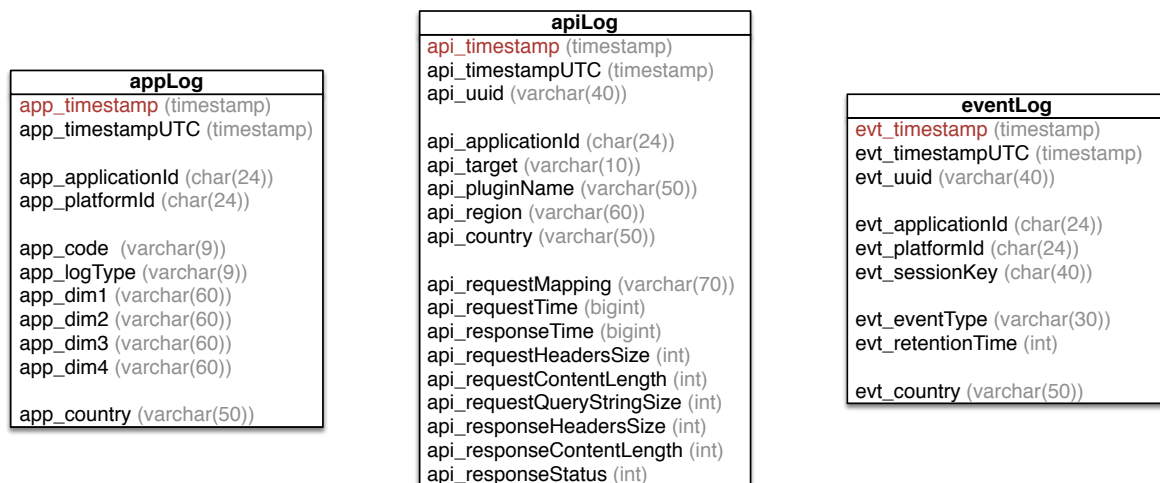


Figure 36 - Amazon Redshift table model

¹ The price changes whether the cluster is reserved for 1 year, 3 years, or not at all (on-demand use: the cluster can be shut down at any time and costs only when it is running). The costs for 1 year and 3 year are calculated with reserved instances.

² Loading data into Redshift in a simple and efficient way requires the use of Amazon S3 buckets. The calculated cost includes 160 GB of storage per month, 1,000,000 PUT and 1,000,000 GET requests of files in a bucket.

Some others field values were adapted before getting inserted. Amazon Redshift does not support time zones for instance. However, Appgrid is used by applications worldwide and the aggregation of events by day should be done for a day in the application's time zone, and not any other arbitrary time zone. For this reason, the consumer process in charge of the insertion of events into Redshift will store two timestamps: one in Universal Coordinated Time (UTC), which is the server's time, and one in the application's time zone. This greatly simplifies the queries made on the data stored in the system, while relieving it from having to shift the timestamps each time a query is to be made.

Also, data denormalization¹ operations are performed on the events pushed into the database. The timestamp shifting can be seen as one of them, but some redundant information is stored as well so as to eliminate the need of *JOIN* operations in queries.

There are also some specific cases and fundamental differences between MongoDB and typical relational databases: document-based databases like MongoDB can store document with any number of fields, and of any type, while relational databases like Amazon Redshift have an immutable field set per table. Document can have subdocuments and store nested arrays of objects, while table-based databases have a flat architecture, and therefore require data to be “flattened” before being inserted. That is also a task done by the consumer process.

7.2.3 Aggregation processor

In the original architecture, the aggregator processor, in charge of querying the data it needs to crunch to generate the desired output was the MongoDB replica set itself. In the architecture proposed in this section, a new Java application, completely independent from the others instances, is responsible for connecting to Amazon Redshift, issuing the required SQL requests, wait for the results and then fill the appropriate MongoDB collections with the aggregation results.

This new Java application has been developed in a modular way: new aggregation queries can be easily added to the system. It is a standalone and self-sufficient application, which can run in a multithreaded mode, to aggregate in parallel Event, Application and API raw logs. The application also supports bulk *upsertions*² of aggregation results into MongoDB to make the overall process even more efficient. The whole process can be controlled with arguments when invoked. In particular, the period on which the data should be aggregated can be either *loosely* (meaning that the process will automatically expand the provided time boundaries if needed to compute reliable data. For instance, if the boundaries span only over one day, the aggregation process will extend this to the enclosing week and month to aggregate the required data set for each aggregation granularity) or strictly specified (strictly will force the aggregation between the specified limits, that will lead to inaccurate aggregation if the period is not strictly one-month long). Specifying a strict limit may though be interesting for testing purposes.

¹ See glossary page 88.

² See glossary page 88.

7.3 Performance testing architecture

To test the overall efficiency of the proposed architecture, the test environment depicted on Figure 37 has been set up.

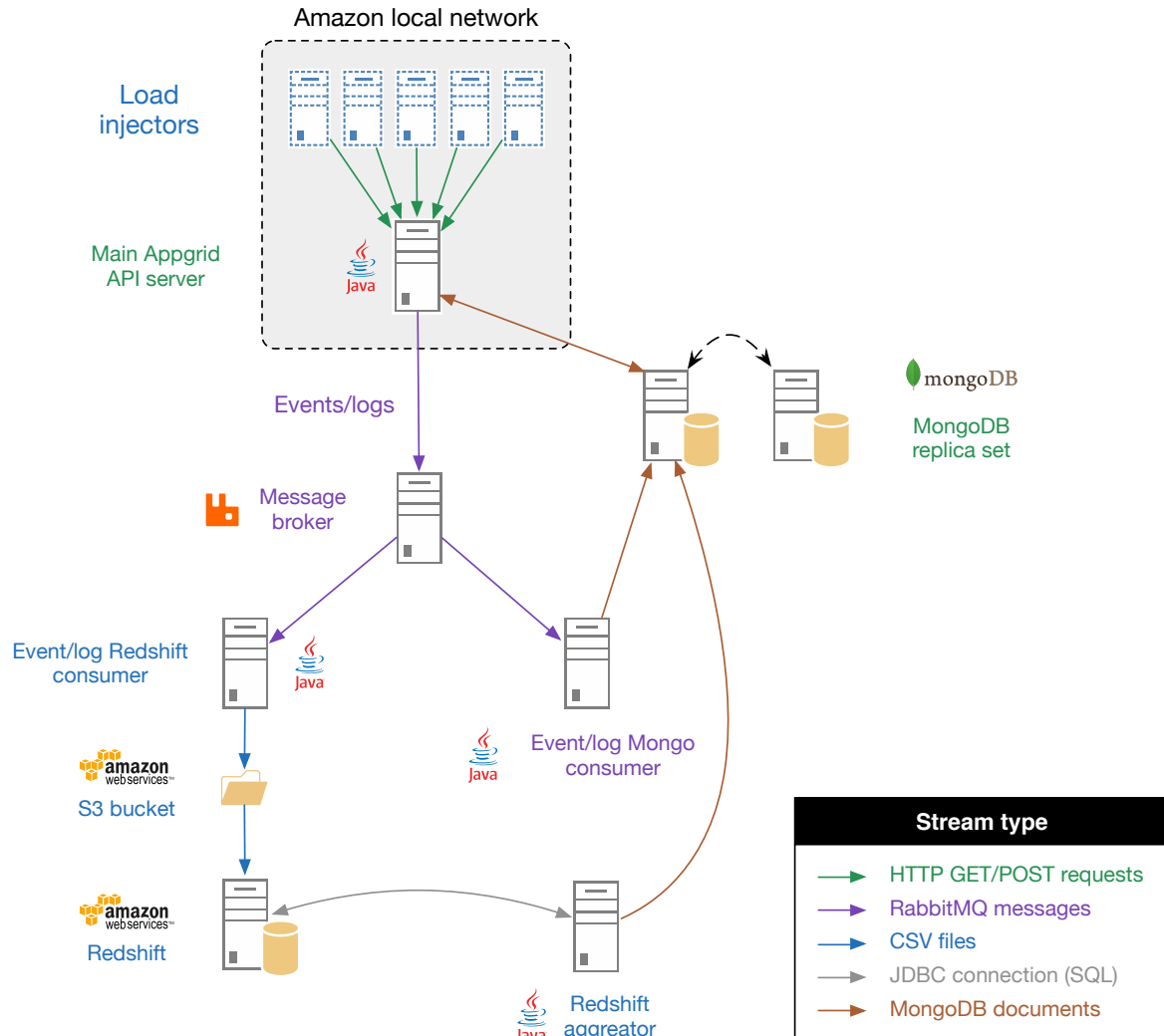


Figure 37 - Appgrid and Amazon Redshift test environment

The same tools for injecting the load toward the Appgrid API server as the ones in section 5.5 were used (i.e. Tsung). Five low-end load injectors were used here, and contrary to the tests carried on in section 5, they were located in the same local network as the server itself, therefore removing any network latency.

Moreover, the scenario played by the load injectors was far more complicated and thorough than the one that was used in section 5 : the load injectors were in fact simulating real users, doing real and *meaningful* actions toward the API server. Different scenarios were randomly chosen each time a new virtual user was started by Tsung. Those scenarios always included the request of a session key (some server spamming was also simulated, with users requiring multiple session keys within a short period), and then were designed to test all the supported requests that one can make toward Appgrid. Random simulated think times (i.e. pause between the requests) were also introduced to reproduce the behaviour of real users.

Therefore, the load generated here was highly similar to the one that can be expected in production. The load profile was very simple: during the course of a test, 15 users were started each second, which is a value that was selected after testing some of them and the one that would keep the server in a stable state. However, this value does not really mean something as is, since different users run different scenarios, and the scenarios themselves have variable duration due to the random think times included between each requests. These tests shown that one API server could roughly support 3,000 simultaneous “real-world” users.

Nevertheless, this test architecture has proven its efficiency and allowed the generation through the real Appgrid API of a large number of events to be stored in the database. This is perfect for testing the consumers and aggregators, to see if all the events that the API may produce are correctly parsed and integrated.

One can notice on Figure 37 that the log consumer for Redshift sends CSV files toward an Amazon S3 bucket, which in turns feed the Redshift database itself. This is one of the possible ways to efficiently insert data into Redshift tables [86]. Amazon Redshift is capable of receiving *INSERT* SQL commands, but it is almost only possible for compatibility reasons and should not be used to stream data into the database. Instead, Redshift is really optimized to import multiple files (e.g. CSV) simultaneously. Having the message broker in the architecture allows the consumer to easily implement local batching of records, just as it was done for the MongoDB consumer (see section 5.2.2 page 44). The queue is just way larger (for the tests, its size was set to 30,000, split into 3 CSV files of 10,000 lines each then), and so is the flush timeout (set to 20 minutes).

Each time that a message is received, it is deserialized, modified as needed (timestamps, etc.) and then serialized in CSV and appended to one of the files constituting the queue. When the queue is flushed, the file descriptors are closed and each one is individually compressed (using GZIP). A manifest file (a JSON file telling how many files were compressed and their locations) is generated and both the manifest and the zipped CSV files are put on an Amazon S3 bucket¹. A JDBC connection is then established between the consumer and Redshift, and a *COPY* SQL command is issued, telling the database to fetch the manifest file, and import what is described inside. Once the import if finished, the modified table is *vacuumed* and *analysed* (two operations required to keep the performance of the database at its best level), and the messages are acknowledged to the RabbitMQ broker.

¹ See glossary page 88.

7.4 Test configurations and checking process

The goal of this section is to demonstrate the efficiency of the new aggregation processor, using Amazon Redshift, compared to the original one using only the data stored into MongoDB. For the test to be meaningful, the two databases should have the same raw input logs, and produce the same aggregated output. The message-oriented middleware comes in handy there, and will guarantee that both MongoDB and Redshift will store the exact same number of logs.

After running a load injection test, all the queues (both the queues on RabbitMQ and the local ones, batching records, on the consumer sides) are emptied and the number of logs in both MongoDB and Redshift is checked and must match. Then, the original aggregation process, crushing the data contained into MongoDB, is launched, and its execution time is measured. Multiple executions were done to make sure that the results were not fluctuating to a great extent.

Once the aggregation process is done, the aggregated data is kept in another database to be compared against the one that Redshift is to produce. The aggregated database is then wiped and the new aggregator using Redshift is launched (to aggregate data on the same period as the original aggregation process). Its running time is recorded and its output is compared with MongoDB's aggregation output to confirm that they match and provide the same results.

Note that the records were spread on a date period of about one week. Each time the aggregation processes were launched, they were asked to aggregate data for the whole month from which this week was part of. No other records were in the databases for this specific month.

7.5 Test results and conclusions

Table 11 presents the results of the tests. Note that the new (or Redshift) aggregator process was run in a single-threaded mode, to be able to compare its results “fairly” against the original aggregation process. The multi-threaded process is compared to the single-threaded one in Table 12.

	Limited data set			Large data set		
	# records	Aggreg. time [s]		# records	Aggreg. time [s]	
		Original	Redshift		Original	Redshift
EventLog	1,934,283	3,420	7	4,486,164	9,853	12
ApiLog	6,044,317		5	14,355,970		16
AppLog	1,218,124	60	28	2,960,627	Failed ¹	52
Total						
(Diff/original)	9,196,724	3,480	40 (-99%)	21,802,761	9,853²	80 (-99%)
Records aggregated/sec	/	2,643	229,918	/	2,213	272,535

Table 11 - Aggregation time comparison

¹ The aggregation output was too large to be handled correctly by the aggregation system (exceeded MongoDB's maximum document size of 16 MB). The results assume AppLog's records as aggregated in 0 second.

² Almost three hours.

The original aggregation process is not clearly separated on a “per-collection” basis when it aggregates the data. It is therefore impossible to differentiate the time the process spends aggregating documents from *EventLog* and from *ApiLog*. The sum of the times spent by the process on each is used instead.

The results are impressive: while on the limited data set, the original aggregation process required almost an hour to generate all the figures needed, the Redshift equivalent was done in less than a minute. The number of records processed per second is 100 times larger than the one that can be obtained with MongoDB Map-reduce jobs.

On the large data set, the results are still very good for Redshift: the number of records aggregated per second increased by almost 20% compared to the limited data set, while MongoDB's performance decreased in approximately the same proportion. The MongoDB's values are also underestimated (namely, the total time spent for aggregation, and the records aggregated per second rate), as the applications logs could not be properly aggregated. The performance of the original aggregation process appears then to be slightly better than what it is in reality.

7.5.1 Single vs. multithreaded aggregator process

Those results show how interesting it is to populate two specialized databases with almost the same data at the same time. But the results can be even better in the Redshift case when used in multithreaded mode. Redshift is indeed capable of running up to 50 queries simultaneously. By default however, the number of simultaneous queries is capped to 5, and this setting was kept for the tests. 3 connections, one for each table, were opened toward Redshift by the aggregation processor in multithreaded mode.

Table 12 compares the performance of the Redshift aggregator, whether it is run in single or multithreaded mode. The data sets used for these tests are the same as those used for the results presented in Table 11.

	Limited data set		Large data set	
	Aggregation time [s]		Aggregation time [s]	
	Single thread	Multi thread	Single thread	Multi thread
EventLog	7	12	12	29
ApiLog	5	10	16	23
AppLog	28	22	52	53
Total	40	22	80	53
Records aggregated/sec (Diff/single-threaded)	229,918	418,033 (+81%)	272,534	411,372 (+51%)

Table 12 - Single/multi-threaded Redshift aggregation process comparison

Aggregating simultaneously the data almost halves the duration of the aggregation process compared to the single-thread version with data sets that are limited. This really makes the process suitable for near real-time aggregation.

Note however that this architecture does not deserve the “real-time” data aggregation label, as there are multiple points in which the data can be stored and buffered: in the broker, where messages can be stored for an unlimited period (if no consumer can process them right away), as well as in the consumers, where they are buffered to be inserted in batches into either MongoDB or Redshift. The consumer in charge of filling the Redshift database is flushing its object queue every 20 minutes at least. This value can be changed in the consumer process of course, but seemed in this case to be a good balance between efficiency and costs¹.

Therefore, with this setup and these settings, aggregation can be achieved with a 30-minute delay, with the highest accuracy possible: the aggregation process does not perform simplified computations as it may be the case in some other real-time aggregation architecture like the Lambda architecture and its real-time layer. They are finite, based on data really collected, and providing the data aggregated in the way it should be [87], [88, Sec. Speed layer]. Moreover, the architecture is not made more complex than it needs to be: aggregation is done using only one tool, and queries are written only once, in a unified language.

On large data sets, the difference between single and multithreaded versions is limited because most of the time is not spent in issuing and calculating the results of the queries, but rather inserting their results into MongoDB. This is especially true with the aggregation of AppLog: more than 500,000 documents² have to be inserted into the collection storing the results of aggregation about application logs. This process is highly time-consuming and limited by the performance of MongoDB. According to the Redshift monitoring console, the two queries that aggregate application logs are executed in less than 10 seconds, the rest of the time being used to extract the results and push them to MongoDB.

7.5.2 Disk space usage

One interesting fact about Amazon's Redshift is the efficiency of its compression algorithms. Data can in fact be compressed when stored into the tables, without affecting performance. Table 13 and Table 14 present some bare results.

MongoDB disk usage	ApiLog	AppLog	EventLog	Total
Number of records	77,662,075	363,881	30,527,012	108,552,968
Total size ³ [MB]	98,587	381	24,965	123,932
Size of 1,000,000 records [MB]	1,270	1,046	818	3,134

Table 13 - Space disk usage by MongoDB

¹ Pushing data into Redshift is not free, while querying it is. The more data the consumer sends in one batch, the lower the bill. See section 7.2.1 page 72.

² To be exact, 2x250,000 documents, in two different aggregated collections. This number is related to the way the load is generated: load generators create pseudo-random logs with a log code randomly picked between 0 and 99999, and on which the logs are aggregated every day on a per-application basis.

³ The sizes of the indexes are included in the total size for each collection.

Amazon Redshift disk usage	ApiLog	AppLog	EventLog	Total
Number of records	16,826,708	2,970,197	6,856,435	26,653,340
Total size [MB]	640	136	480	1,256
Size of 1,000,000 records [MB]	38	46	70	154

Table 14 - Space disk usage by Amazon Redshift

Redshift compression clearly does a good job, being more efficient than MongoDB, which is impacted to a great extent by the size of the indexes it must maintain to be able to perform queries in a reasonable time. Redshift does not need those indexes and, combined to the efficient columnar storage and compression scheme it uses, can store more than 20x more data than MongoDB for the same disk space usage, according to these specific tests.

The smallest Redshift node available has 160 GB of storage space available. Therefore, more than 1 billion records can be stored in the available disk space of just one node. If one user session is said to generate 30 records on average (see section 7.5.3 below), it appears that one Redshift cluster could store more than 35 millions of sessions. MongoDB would only be able to store 1,7 million sessions with a similar disk storage capacity.

7.5.3 RabbitMQ and Amazon Redshift resiliency

One side note about the resiliency of the system: one of the advantage of using message-oriented middleware is that provided that the broker is kept up and running, applications should not be affected in anyway by a failure of the database or the consumers. The broker queues messages and flushes them once the failure is solved. A quick failure simulation (done by shutting down all the consumers during four days, while still using the API, and thus filling the queues) revealed how handy and efficient it can be. Results of this side test are presented in Table 15.

Number of queued messages (all queues ¹)	9,874,310
Time to complete flush	1,290 s (21 min 30 s)
Combined average message flush rate/s	7,634

Table 15 - Failure recovery test results

¹ The reader should remember that each message is duplicated when it enters into the broker. Therefore, the API effectively sent *only* 4,923,655 messages.

The results are very good, by considering the fact that a single consumer was used, consuming the data from 6 different queues (two per each collection/table to fill, one in MongoDB, one in Redshift). The complete catch up occurred in less than 30 minutes, with absolutely no downtime or data loss on the API side. If a session is considered on average to generate 30 (before duplication) messages¹, this very simple architecture, with a unique consumer, was able to absorb more than 160,000 user sessions in 21 minutes, or 7,630 sessions per minute. Note also that with the batch sizes presented in section 7.3, all the queues were on average emptied at the same rate, whether the data was inserted into Redshift or MongoDB.

7.5.4 General conclusion

The architecture proved its efficiency, and tools like Amazon Redshift, with their linear scaling scheme (the number of records aggregated per second is even getting better as the set of data grows, but the results are a bit biased as MongoDB's operations are also involved to store the aggregation results and do not only reflect Redshift's performance), can help a lot when millions of rows need to be processed. Profiling the Java application issuing the queries and storing the results may lead to additional performance enhancements, while designing differently the collections in which the aggregation outputs are stored may also speed up the overall process to a great extent.

¹ Which is the double of what is currently seen in production: on average, 2 entries in EventLog, 12 entries in ApiLog, while the samples used to determine these values did not include any information about AppLog.

8 Conclusion and future work

This section aims at summarizing what has been explained in this report, formulating some quick guidelines and proposing new perspectives to carry on with this work on message-oriented middleware, to, as the title of the thesis suggests, contribute to build strong and scalable big data analytics architecture.

8.1 Advantages and perspectives offered by message-oriented middleware

As described in section 2.3, message-oriented middleware give the ability to system architects to rethink the way asynchronous operations are done. By externalizing those processes, it gives them freedom to really defer operations and relieve the main applications servers. Applicable operations include log and event management, e-mail and other contact jobs, and more globally all one-way asynchronous communication between two processes or instances.

The producer-broker-consumer *triumvirate* enforced by message broker systems may definitively be the way to go to save resources on the main application instances, distribute the work across multiple servers while keeping future in mind: those systems are inherently scalable, as more producers-consumers can be connected to the broker as the number of messages to handle increases. The two tested systems – Apache Kafka and RabbitMQ – provide replication, clustering, fault tolerance and load balancing capabilities to make the broker itself scalable as well.

Consumer processes can be optimized as well to constantly take advantage of features offered by applications that are typically used only in very specific cases. Bulk insertion in databases, batch pre-aggregation of data on low scale, consumption throttling either by autosensing the health of the manipulated instances or by administrators... Architectures relying on messaging middleware are not so new but they open new interesting perspectives in term of operational management as well: downtimes and maintenance of consumers or databases can be completely (up to a certain point) transparent for the end user thanks to the message queuing system. Traffic bursts can be also absorbed by these systems, without endangering the whole architecture.

One very nice advantage, as presented in section 7 , which may be eventually the main reason leading to the adoption of such systems in big data workflows is the fact that multiple consumers can, from a single message posted by the producers, process it in different ways. Depending on the broker queue architecture, the process may involve message duplication on the broker, but the principle in itself is really powerful, as shown by the example given on Figure 38.

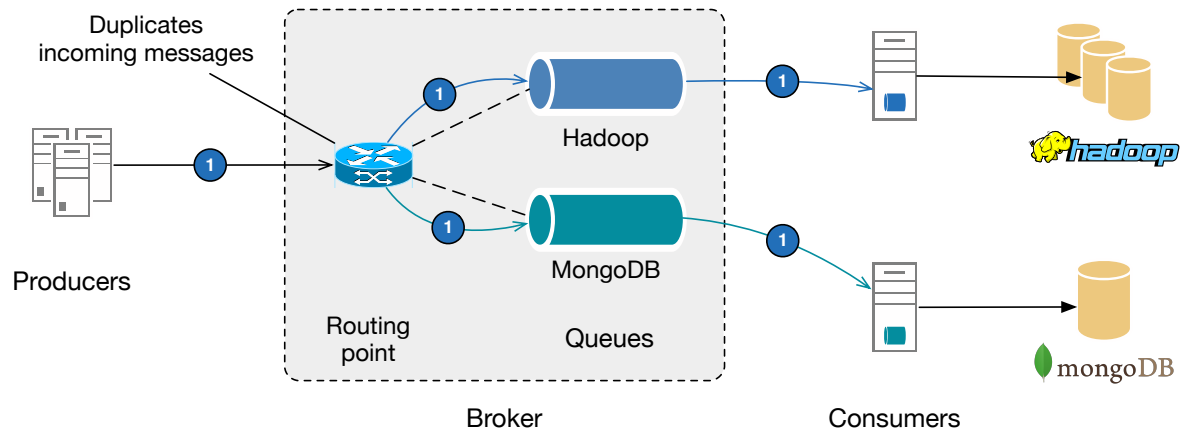


Figure 38 - Multi-consumer/task architecture

The same message that originated at a producer would be processed two times, by two different processes, at two different times and paces, to be inserted into two different data storage spaces (here, MongoDB and Apache Hadoop [89]). This kind of architecture is really powerful, in that it allows multiple data storage software to be used at the same time (using one main traditional MongoDB cluster for individual record access, and an aggregation, analytics-oriented system like Apache Hadoop or Amazon Redshift to perform queries on huge number of lines [77]), without requiring one specific data storage space to be used as the source for the others. In other words, the main data set is the messages, not any sort of database. As such, the additional database systems that may be used would not query the others for data they can get from messages. Thus relieving them from thousands of avoidable queries.

This multi-consumer possibility combined to all the advantages presented before confirms the high potential of the usage of message-oriented middleware to build efficient, rational and scalable data management and analytics architectures.

8.2 RabbitMQ vs. Kafka

The tests presented in this report only apply to a specific architecture, and show trends more than absolute figures. Two systems (RabbitMQ and Kafka) were tested among all of those available on the market, and as such, they are far from being exhaustive enough to affirm that one of them is the best system to use.

However, some conclusions and guidelines can be drawn from them. The primary reason that would make one prefer Apache Kafka to RabbitMQ is the need of complete and relatively long-term data persistence. As shown during the tests, Kafka is really the way to go if queue data should be stored onto the disk, and performs way better than RabbitMQ in this use case. If high rates of messages are expected and if one is afraid of losing data because of some broker failure, then Kafka provides a proper level of protection regarding data integrity and availability.

Still, there are some major disadvantages in Kafka, and RabbitMQ is way better in almost all of these topics. The first one is the way Kafka systems are set up, managed and monitored. As mentioned in section 4.2.3 page 33, managing a Kafka instance is clearly not the simplest thing, as the software is not conveniently packaged, is not self sufficient (need for an Apache ZooKeeper instance) and is absolutely not easy to watch. Exposing MBeans does not make system surveillance easy.

On all those points, RabbitMQ is really more attractive: well packaged for most architecture, completely self sufficient and providing a handy and convenient web management and monitoring interface... Contrary to Kafka that looks like a fresh, nearly experimental product (which it was, but it has now left the Apache Incubator and should therefore be more of a final product right now), RabbitMQ has the tools and features that make it “professional”. Documentation and examples are often messy or mistaken for Kafka, just as debugging problems is. During the tests, very annoying and undocumented problems happened, with unexpected exceptions (some of them being even present under normal conditions and should be disregarded in the logs [90]!) in cases in which such systems are expected to give more resilience against faults (e.g. killing either the broker or the consumer sometimes led to impossible situations, requiring the broker to be wiped or the system to be restarted to reactivate things).

For all those reasons, RabbitMQ appears to be the tool to go, the only one that is reliable enough to be said production-ready. Apache Kafka has a great potential, is used by some very big companies like LinkedIn (company at which it originated [42]), but is too young and has a too limited community behind for a company like Accedo to rely on such a product. RabbitMQ (via Pivotal, its editor) provides enterprise-grade support if needed, as well as talks, training programmes and has an extensive documentation and user base.

The tool worked the way it should, and should there be one tool to be recommended for implementing message-oriented middleware in an existing architecture, this would be without too much hesitation RabbitMQ.

8.3 Event-oriented and service-oriented architecture

As discussed in section 2.2 page 13, event-driven architecture (EDA) or even staged event-driven architecture (SEDA) are becoming popular and may rely on the results of this study to be implemented with the right tools [13], [14]. The principle behind them is to do processing asynchronously as much as possible, and distribute the work across entities that would react to events to which they subscribe.

In this way, one possible breakthrough change may be to test a completely isolated implementation of the producer, which would disconnects it totally from the database (both for read and write operations) when it matters event and log posting. In this way, the main API server would be all the more responsive, as the only task it would have to do would be to send back an OK HTTP code to the client while pushing a message to the broker. All the tasks would be deferred to the consumers.

As attractive as this solution seems to be, it still has some major disadvantages. Queries are right now queued only if they are valid and linked to a valid session key. Here, the validation would be done asynchronously, while a 200 HTTP code would be returned to the client, even if the query were to be incorrect. Moreover, it would generate more load than necessary on the broker and the consumer, especially in case of Denial of Service (DoS) attacks, as faulty messages would get queued and processed even though they were incorrect. With a Kafka-like broker, it would also irremediably alter the “master copy” of the data it is meant to store.

Along with those problems come code duplication and query multiplication (if processing should be done, then all the different consumers consuming the same message set, each receiving a copy of all messages, are likely to do the same queries). Considering also the fact that for backwards compatibility reasons, the query should return to the client some information fetched from the database, it may not be the easiest and cleverest modification to do. Moreover, while writing to a MongoDB replica set requires a communication with the primary server only, data can be read on all the members of the replica set, and therefore the load can be spread across the set (with some concerns about data freshness though [91]).

Nevertheless, message-oriented middleware may help implementing such architecture, using some sorts of “chained queues”, as depicted on Figure 39.

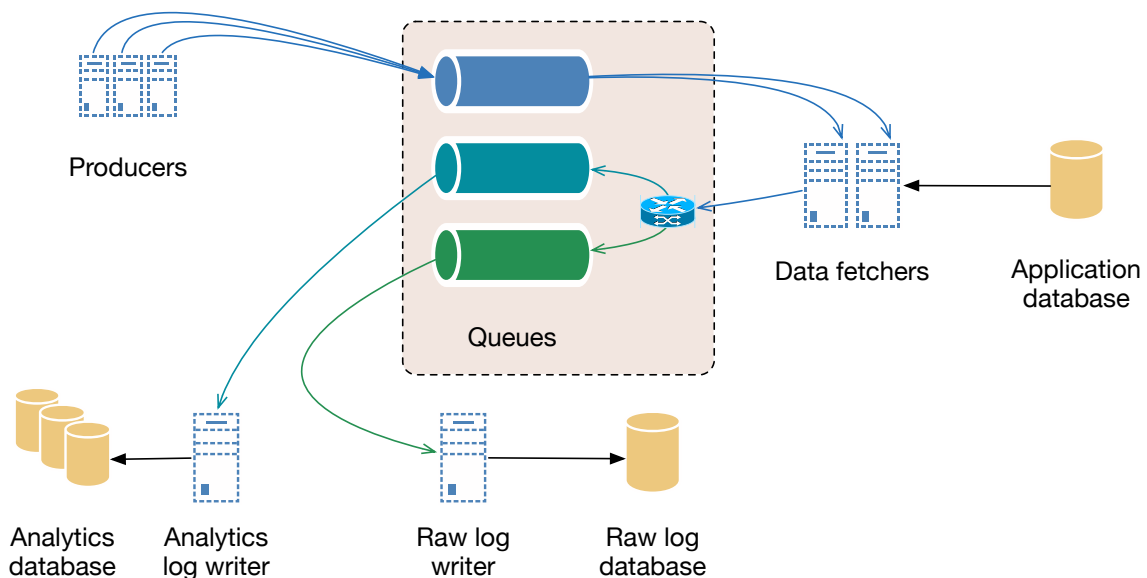


Figure 39 - Distributed and progressive data processing using chained queues

Note the arrow orientation between the databases and each consumer/producer process. The principle here is that initial producers push basic events in a queue, without much information inside the message. Then, a first consuming process will “hydrate” the event and append to it additional information required by future processes in the application database. Then, the messages are pushed back to the broker and duplicated in two queues to be inserted into two different kinds of databases. This architecture brings all the best things discussed in this section: initial producers are disconnected from the application database; events are progressively “augmented” as they flow through the queues, and everything is done asynchronously. Moreover, no code duplication is needed, and queries for elements that are shared by processes at the bottom of the processing chain are made only once by those at the top.

This kind of successive processing may also be seen as pure event processing and take advantage of event database systems like Event Store [92]. Messages would be directly store in this database instead of queues, and event would trigger actions on consuming processes. Those kinds of architectures are also closely related to the micro-service and service-oriented architectures (SOA) that are getting more and more popular [93, Ch. 6], [94, Ch. 4], [95]. Using a message-oriented middleware to have the different services communicate seamlessly really helps to build such an infrastructure, instead of relying on RMI or HTTP, which induces for the latest a non-negligible latency on each query, and does not provide any built-in queuing facility.

8.4 Future work

All the following assumes that a message-oriented middleware has been selected and implemented in the architecture. RabbitMQ is the one discussed hereafter. Performance tweaking of the broker's settings should be performed, where applicable. Different prefetch count values and their impact on the performance should be tested. Clustering and load balancing should also be tried.

Multiple consumers (as Figure 38 page 83 suggests) should be added to make sure it does not impact badly the performance of the broker, and concurrent consumer processing data from the same queue can also be tested (the easiest way to scale systems horizontally). While testing, extreme care should be taken in monitoring the database systems on which producers and consumers may access. Databases may indeed get overwhelmed by the number of highly optimized processes that would connect to it.

For RabbitMQ with persistence enabled, one identified bottleneck was the I/O throughput capacity of the machine. Retrying the test on a disk-optimized instance may show better performance and maybe compensate the real performance trade-off that security brings in RabbitMQ as shown in the tests that this report presents.

A large range of potential optimizations can also be carried out in the code of both producing and consuming processes. Producers may try to simulate batch delivery, while tweaking the bulk insertion parameters (number of records to write simultaneously and timeout value) should bring some additional performance improvements.



But the best thing to do next is to take advantage of this architecture to feed multiple database systems and try many of those (even directly in production, as consumers can be plugged in and out without any interference on other processes!), to see which one is the best when aggregating large number of records. Section 7 presented Amazon Redshift and its power, but there are other tools on the market like Google BigTable that can perform similar tasks. Comparing them using message-oriented middleware is easy and would allow to build the most efficient and scalable data analytics architecture.

9 Glossary

*This glossary provides quick definitions to most of the unusual terms used in this document. More details can be found in the references. **Please note that this glossary has multiple parts, to separate the vendor-specific words from the general ones.***

Consumer independent process that consumes data (i.e. *dequeue* messages from the broker).

Common Table Expression (CTE) SQL feature that allows one to write queries that use as input the results of one or more specified, temporary queries. *WITH* introduces CTEs in SQL queries.

Data denormalization technique to improve the performance of read operations in database that relies on data duplication and grouping (e.g. instead of a *Customer* table with a *town* attribute and an *Order* table with a link to a *Customer*, if the only information needed is the town, then the *town* field may be duplicated in all the *Orders* rows to avoid a costly *JOIN* operation) [96].

Interface Description Language (IDL) Language used to specify the structure of objects to build components that can deal with them, no matter their programming language. For instance, CORBA is an IDL, and RMI is based on it.

Map-reduce Map-reduce is a programming paradigm originated at Google that separates data aggregation processes on large sets in two parts: one mapping operation, in which the set is filtered to keep the data to aggregate, and one reduce operation, in which the remaining rows are summarized [97].

Message raw string of bytes, sent by a producer to a broker together with a simple header, that is queued and then delivered to one or more consumers, that are programmed to deserialize the message's content and process it.

MongoDB MongoDB is a database system that stores "documents" in "collections". This is a NoSQL database: a document is a BSON (Binary JSON) entity and is the equivalent of a row in a SQL database, while a collection is like a table. It is made to be distributed across multiple machines, and does not enforce any fixed table/collection scheme. It supports Map-Reduce jobs for complex data aggregation operations on large data sets [98].

ParNew GC ParNew is a garbage collector (GC) that uses multiple GC threads to do the work [99]. ParNew is a "stop-the-world" GC, which means that the application becomes unresponsive during its execution.

Persistence property of queues and messages that survive a software failure or restart (i.e. they are written to the disk).

Producer process that will send messages to the broker. They can be synchronous or asynchronous (i.e. wait or not for having more messages to send to batch them and/or acknowledgements).

Publish/subscribe messaging messaging scheme in which multiple consumers can consume the same message, at different rates [2, Sec. 1.3.2]. By opposition to queuing messaging.

Publisher **confirmation**

acknowledgement sent by the broker to the producer process of a message when it would have successfully enqueued it.

Queuing messaging (or point-to-point) messaging scheme in which only one consumer listening to a queue will have the opportunity to receive a specific message sent to this queue [2, Sec. 1.3.1]. By opposition to publish/subscribe messaging.

S3 (bucket) Amazon's file storage system in the cloud. A bucket is a space (like a folder) in which files can be placed (or *put*) and retrieved.

Serialization process that converts a data structure to something that can be either stored or transferred (usually, an object converted to a string) [100]. JSON is a common serialization format.

Upsert database operation that is made of a query *Q* and a document to *upsert D*, that results either in the insertion of the document *D* into the database if the query *Q* returns no result, or updates the returned records with the values of document *D* if the result set of query *Q* is not empty. It is therefore a “create or update” operation, or “update or insert”, contracted as *upsert*.

Kafka specifics

Acknowledged message state of a message that has been successfully enqueued by the leader of a given partition but not replicated yet on its replicas.

Asynchronous producer producer that does not wait either for a message to get acknowledged or committed to move on and send another message. No guarantee is then provided regarding the enqueueing status of the messages in case of producer failure (before they are sent and acknowledged).

Broker logical instance of Kafka that can receive messages and redistribute them. Talks with one Zookeeper server, and store one or more partitions.

Committed message state of a message that has been successfully enqueued by the leader of a given partition, as well as successfully replicated on all its replicas (if any). Refer to “in-sync”.

Consumer process that consumes data of one or more partitions. It sends to the leading broker the offset from which it wants data and receives it.

Consumer group represents a set of consumers that usually do the same task. Consumers tag themselves to which consumer group they belong. Zookeeper keeps consumer groups' namespace globally. Only one consumer in a consumer group receives a message from a specific partition. Each message will be sent to one consumer in every consumer group that exists.

In-sync status of a set of Kafka brokers that are replicas of a given partition that means that all the replicas have the same data as the leader for this partition.

Leader when multiple brokers exist, partitions will be distributed across a set of brokers (distribution automatically chosen or manually given). One will act as the leader (i.e. producers will send messages to it, and consumers will read data from it) and the others will act as replicas. The leader replicates the data to the replicas.

Offset "identifier" of a location in a partition's queue. Each message has a different (increasing) offset, and consumers can query specific offsets to (re)get all the data they need to catch-up.

Partition represents a *sub-queue* of a topic, that starts at a specific offset and that is then filled with data sent by producers to this partition. For efficiency, producers can decide to put data in a specific partition, or use any kind of load-balancing process to fill the partitions in a clever way. A partition stores messages in the order it receives them. This is the smallest entity that Kafka brokers can replicate.

Replicas when multiple brokers exist, a partition may be replicated on different brokers to improve resiliency. Each partition has a leader broker and a set of replicas (if any available).

Replication factor number of brokers on which every partition will be replicated. 1 then means no replica; just the "leader" holds the partition.

Synchronous producer producer that waits either for an acknowledgment or a commit notification for each message it sends. Guarantees that messages have been enqueued.

Topic a set of partitions, constituting a queue.

RabbitMQ specifics

Binding a queue has to be bound to one or more exchanges to receive messages. Bindings connect queues and exchanges, with an optional routing key that can be used to specify conditional routing rules. The default exchange (which has an empty string for name, and is a "direct" exchange) is automatically bound to all the queues defined in the broker with the name of the queue as routing key. It means that messages sent to the default exchange with the name of the queue as routing key will get delivered to this queue (and only this one).

Consumer process that runs independently and that will consume data. The consumer may be either passive: once connected to the broker, it will receive data without doing anything if using the push API (the broker sends messages when needed), or will ask regularly the broker for data with the pull API.

Consumption acknowledgment requires consumers to send an acknowledgment to the broker for each message that has been successfully processed. A message may then be deleted on the broker only if it has been acknowledged. If not and the connection between the broker and the consumer gets broken, the message is assumed lost, *requeued* and then retransmitted to another consumer.

Durability RabbitMQ name for persistence. Can be applied to exchanges, queues and messages.

Exchange entry point for messages. Producers push messages to a specific exchange which then delivers them to zero or more queues according to their routing key, the exchange type and the bindings the exchange has. Exchange types include "direct" (i.e. received messages whose routing key is equal to one of its bindings' keys are delivered to the queue linked to this binding), "fan-out" (i.e. messages are pushed to all the queues this exchange is bound to), and the more advanced "topic" and "header" [52].

Queue space in which messages will eventually end (if not dropped) before being transferred to consumers. Can be tagged as "durable" (persistent).

Prefetch count number of messages a broker can send to a consumer without receiving any acknowledgement [101]. Can be seen as the window size in TCP exchanges.

10 List of figures

Figure 1 - Accedo Appgrid solution overview	8
Figure 2 - Overview of a message-oriented architecture.....	12
Figure 3 - Full-duplex communication between processes	13
Figure 4 - Example of a scenario with duplicated messages	16
Figure 5 - Current Appgrid architecture	19
Figure 6 - Original log and event insertion process (simplified).....	21
Figure 7 - Message-oriented middleware in Appgrid architecture	22
Figure 8 - Basic message-oriented Appgrid log insertion process	23
Figure 9 - Optimized message-oriented Appgrid log insertion process.....	25
Figure 10 - Logo of various message-oriented middleware or providers (trademarks).....	26
Figure 11 - Global Apache Kafka architecture (with one topic, one partition, replication factor 4)	28
Figure 12 - Architecture of a topic	29
Figure 13 - Consumer group concept (example with one topic and one partition).....	30
Figure 14 - Replication example (replication factor of 2, 3 partitions, 1 topic).....	31
Figure 15 - Simplified overall RabbitMQ architecture	34
Figure 16 - Direct exchange example (routing keys = queue names = colours).....	35
Figure 17 - Fan-out exchange example (routing key meaningless, three queues, two bindings only)	35
Figure 18 - RabbitMQ clustering with queue mirroring policy set to <i>all</i>	37
Figure 19 - Class diagram, <i>tv.accedo.appgrid.data.bus</i> package.....	43
Figure 20 - Thread architecture of consumer process class diagram	44
Figure 21 - Message processing classes of consumer process class diagram.....	45
Figure 22 - Message structure and serialization logic.....	49
Figure 23 - Kafka topic architecture	50
Figure 24 - RabbitMQ queue architecture	51
Figure 25 - Architecture used for load testing.....	52
Figure 26 - Test transaction.....	53
Figure 27 - User load that should be generated vs. time.....	54
Figure 28 - Comparison of the global message push rate between Kafka in synchronous mode and RabbitMQ with persistence	63
Figure 29 - Total number of messages (in queue + not acknowledged by consumer) vs. time, for RabbitMQ with persistence	64
Figure 30 - Broker CPU usage (for Kafka) vs. time	65
Figure 31 - Broker CPU usage (for RabbitMQ) vs. time	66
Figure 32 - CPU usage by consumer process (with RabbitMQ non-persistent).....	67
Figure 33 - Benefits of bulk insertions into MongoDB vs. single insertion (using RabbitMQ without persistence)	67
Figure 34 - General aggregation process	69
Figure 35 - Message-oriented middleware with Amazon Redshift.....	71
Figure 36 - Amazon Redshift table model	73
Figure 37 - Appgrid and Amazon Redshift test environment	75
Figure 38 - Multi-consumer/task architecture	83
Figure 39 - Distributed and progressive data processing using chained queues.....	85

11 List of tables

Table 1 - General information comparison between Apache Kafka and RabbitMQ.....	39
Table 2 – Broker capabilities comparison between Apache Kafka and RabbitMQ.....	40
Table 3 – Producer (official Java client only) capabilities comparison between Apache Kafka and RabbitMQ.....	40
Table 4 – Consumer (official Java client only) capabilities comparison between Apache Kafka and RabbitMQ.....	40
Table 5 - Test configurations' summary.....	55
Table 6 - Hardware specifications of the server instances used for the tests.....	56
Table 7 - Synthetized test results about the Appgrid API.....	61
Table 8 – Synthetized results about message rates.....	62
Table 9 - Consumer and broker performance comparison.....	65
Table 10 - Amazon Redshift specifications and cost.....	73
Table 11 - Aggregation time comparison.....	77
Table 12 - Single/multi-threaded Redshift aggregation process comparison	78
Table 13 - Space disk usage by MongoDB.....	79
Table 14 - Space disk usage by Amazon Redshift.....	80
Table 15 - Failure recovery test results.....	80

12 References

Some images used in this report may be protected by a trademark and are used for illustration purpose only. Stencils used to create the diagrams and figures of this document are referenced herein [102], [103]. Figures contained in this document may not be reused without prior permission. While most of the references are in English, some might only be available in Swedish or French.

- [1] M. Minelli, M. Chambers, and A. Dhiraj, *Big Data, Big Analytics : Emerging Business Intelligence and Analytic Trends for Today's Businesses*. New York: John Wiley & Sons, 2013.
- [2] E. Curry, "Message-Oriented Middleware," in *Middleware for Communications*, Q. H. Hmoud, Ed. John Wiley & Sons, Ltd, 2004, pp. 1–28.
- [3] M. Kosinski, "With big data comes big responsibility," *Financial Times*, 14-Mar-2013.
- [4] A. Oboler, K. Welsh, and L. Cruz, "The danger of big data: Social media as computational social science," *First Monday*, vol. 17, no. 7, Jun. 2012.
- [5] P. C. Webster and W. Kondro, "Medical data debates: Big is better? Small is beautiful?," *CMAJ Can. Med. Assoc. J.*, vol. 183, no. 5, pp. 539–540, Mar. 2011.
- [6] C. Porter, "Little privacy in the age of big data," *The Guardian*, 20-Jun-2014.
- [7] L. Sweeney, "k-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, no. 05, pp. 557–570, 2002.
- [8] D. Wogan, "Big Data, Big Energy," *Sci. Am.*, vol. 310, no. 1, pp. 22–22, Dec. 2013.
- [9] Karin Svensson, "Borg flörtar med Facebook," *di.se*, 20-May-2014. [Online]. Available: <http://www.di.se/artiklar/2014/5/20/borg-flortar-med-facebook/>. [Accessed: 29-Aug-2014].
- [10] Peter Alestig, "Sänkt skatt ska locka hit it-jättar," *SvD.se*, 20-May-2014. [Online]. Available: http://www.svd.se/naringsliv/sankt-skatt-ska-locka-fler-it-jattar-till-sverige_3577004.svd. [Accessed: 29-Aug-2014].
- [11] S. Flucker and R. Tozer, "Data Centre Energy Efficiency Analysis to minimize total cost of ownership," *Build. Serv. Eng. Res. Technol.*, vol. 34, no. 1, pp. 103–117, Feb. 2013.
- [12] S. Tarkoma, *Wiley Series on Communications Networking and Distributed Systems, Volume 48 : Publish/Subscribe Systems : Design and Principles*. Somerset, NJ, USA: John Wiley & Sons, 2012.
- [13] J.-L. Maréchaux, "Combining service-oriented architecture and event-driven architecture using an enterprise service bus," *IBM Dev. Works*, pp. 1269–1275, 2006.
- [14] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-conditioned, Scalable Internet Services," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2001, pp. 230–243.
- [15] Wikipedia contributors, "Idempotence," *Wikipedia, the free encyclopedia*. 25-Jun-2014.
- [16] Amazon Web Services, "AWS Elastic Load Balancing - Cloud Network Load Balancer," *Amazon Web Services, Inc.*, 2014. [Online]. Available: <http://aws.amazon.com/elasticloadbalancing/>. [Accessed: 11-Jul-2014].
- [17] Pivotal Software, Inc., "25.Task Execution and Scheduling," *Spring Documentation*, 2014. [Online]. Available: <http://docs.spring.io/spring/docs/3.0.x/spring-framework-reference/html/scheduling.html>. [Accessed: 22-Jul-2014].

- [18] MongoDB Inc., “Bulk Inserts in MongoDB — MongoDB Manual 2.6.3,” *MongoDB Manual* 2.6.3, 04-Apr-2014. [Online]. Available: <http://docs.mongodb.org/manual/core/bulk-inserts/>. [Accessed: 11-Jul-2014].
- [19] T. Ouellette, “Middleware safeguards data transactions over the Web. (New Era of Networks Neonweb message-oriented middleware)(Brief Article)(Product Announcement),” *Computerworld*, vol. 30, no. 50, p. 28, 1996.
- [20] T. Ouellette, “Delta relies on IBM’s MQSeries to tie systems. (Middleware Takes Flight) (Delta Air Lines Inc using message-oriented middleware) (Technology Information),” *Computerworld*, vol. 30, no. 49, p. 53, 1996.
- [21] Mike Gilpin, “Capitalware’s 2000 Forecast for the Application Integration Market Page,” *Capitalware Inc.*, 15-Jun-2000. [Online]. Available: http://www.capitalware.com/forecast_AI_market.html. [Accessed: 21-Jul-2014].
- [22] The Apache Software Foundation, “Apache ActiveMQ™ -- Index,” *Apache ActiveMQ*, 31-Mar-2014. [Online]. Available: <http://activemq.apache.org/>. [Accessed: 21-Jul-2014].
- [23] The Apache Software Foundation, “Home - Apache Qpid™,” *Apache Qpid*, Sep-2013. [Online]. Available: <https://qpid.apache.org/>. [Accessed: 21-Jul-2014].
- [24] J. O’Hara, “Toward a Commodity Enterprise Middleware,” *Queue*, vol. 5, no. 4, pp. 48–55, May 2007.
- [25] S. Vinoski, “Advanced Message Queuing Protocol,” *IEEE Internet Comput.*, vol. 10, no. 6, pp. 87–89, 2006.
- [26] Alvaro Videla and Jason J. W. Williams, *RabbitMQ in Action: Distributed Messaging for Everyone*, 1st ed. Manning Publications, 2012.
- [27] Hemant Gaidhani, “vCenter Orchestrator AMQP Plug-in is now Generally Available,” *VMware vCenter Orchestrator Blog*, 16-Aug-2011. .
- [28] Mark Lin, “Get ready to Rocksteady,” *Google Open Source Blog*, 23-Sep-2010. .
- [29] Mozilla, “Mozilla Pulse,” *Mozilla Pulse*, 2014. [Online]. Available: <http://pulse.mozilla.org/>. [Accessed: 21-Jul-2014].
- [30] Simon MacMullen, “RabbitMQ» Blog Archive» RabbitMQ Performance Measurements, part 2 - Messaging that just works,” *RabbitMQ’s blog*, 25-Apr-2012. .
- [31] Adam Bloom, “How fast is a Rabbit? Basic RabbitMQ Performance Benchmarks,” *VMWare vFabric Blog*, 18-Apr-2013. .
- [32] STOMP, “STOMP,” *STOMP*, 2014. [Online]. Available: <http://stomp.github.io/>. [Accessed: 21-Jul-2014].
- [33] Dave Locke, “MQ Telemetry Transport (MQTT) V3.1 Protocol Specification,” *IBM Developer Works*, 19-Aug-2010. [Online]. Available: <http://www.ibm.com/developerworks/webservices/library/ws-mqtt/>. [Accessed: 21-Jul-2014].
- [34] The Apache Software Foundation, “Apollo,” *ActiveMQ’s next generation of messaging*, 2014. [Online]. Available: <http://activemq.apache.org/apollo/>. [Accessed: 21-Jul-2014].
- [35] M. Salvan, “A quick message queue benchmark: ActiveMQ, RabbitMQ, HornetQ, QPID, Apollo...,” *Muriel’s Tech Blog*, 10-Apr-2013. .
- [36] Stuart Charlton, “RabbitMQ vs Kafka: which one for durable messaging with good query features? - Quora,” *Quora*, 12-Sep-2012. [Online]. Available: <http://www.quora.com/RabbitMQ/RabbitMQ-vs-Kafka-which-one-for-durable-messaging-with-good-query-features?share=1>. [Accessed: 16-Jun-2014].
- [37] F. Akgul, *ZeroMQ*. Olton, Birmingham, GBR: Packt Publishing, 2013.

- [38] The Apache Software Foundation, “Apache Kafka,” *Apache Kafka*, 29-Apr-2014. [Online]. Available: <http://kafka.apache.org/>. [Accessed: 10-Jul-2014].
- [39] JBoss, “HornetQ - putting the buzz in messaging - JBoss Community,” *HornetQ*, 16-Dec-2013. [Online]. Available: <http://hornetq.jboss.org/>. [Accessed: 21-Jul-2014].
- [40] J. Kreps, N. Narkhede, and J. Rao, “Kafka: A distributed messaging system for log processing,” in *Proceedings of the NetDB*, 2011.
- [41] The Apache Software Foundation, “Apache ZooKeeper - Home,” *Apache ZooKeeper*, 10-Mar-2014. [Online]. Available: <http://zookeeper.apache.org/>. [Accessed: 11-Jul-2014].
- [42] Jay Kreps, “The Log: What every software engineer should know about real-time data’s unifying abstraction,” 16-Dec-2013. [Online]. Available: <http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>. [Accessed: 05-Jun-2014].
- [43] Martin Odersky, “What is Scala? | The Scala Programming Language,” *Scala*, 2014. [Online]. Available: <http://www.scala-lang.org/what-is-scala.html>. [Accessed: 11-Jul-2014].
- [44] The Apache Software Foundation, “Apache Kafka 0.8.1 - Documentation,” *Apache Kafka*, 29-Apr-2014. [Online]. Available: <http://kafka.apache.org/documentation.html#diskandfs>. [Accessed: 14-Jul-2014].
- [45] Chris Curtin and Thomas Uribe, “0.8.0 SimpleConsumer Example - Apache Kafka - Apache Software Foundation,” *Apache Kafka wiki*, 29-Apr-2014. [Online]. Available: <https://cwiki.apache.org/confluence/display/KAFKA/0.8.0+SimpleConsumer+Example>. [Accessed: 14-Jul-2014].
- [46] Neha Narkhede, “Compression - Apache Kafka - Apache Software Foundation,” *Apache Kafka wiki*, 02-Jan-2012. [Online]. Available: <https://cwiki.apache.org/confluence/display/KAFKA/Compression>. [Accessed: 14-Jul-2014].
- [47] Code Hale, Yammer Inc., “Home | Metrics,” *Metrics*, 2013. [Online]. Available: <http://metrics.codahale.com/>. [Accessed: 14-Jul-2014].
- [48] RabbitMQ, “Launch of RabbitMQ Open Source Enterprise Messaging - Press release.” LShift/CohesiveFT, Feb-2007.
- [49] Pivotal Software, Inc., “RabbitMQ - Messaging that just works,” *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/>. [Accessed: 20-Apr-2014].
- [50] Pivotal Software, Inc., “RabbitMQ - AMQP 0-9-1 Quick Reference,” *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/amqp-0-9-1-quickref.html>. [Accessed: 14-Jul-2014].
- [51] S. Tarkoma, *Standards and Products*. Chichester, UK: Chichester, UK: John Wiley & Sons, Ltd, 2012.
- [52] Pivotal Software, Inc., “RabbitMQ - AMQP 0-9-1 Model Explained,” *RabbitMQ*, 2014. [Online]. Available: <https://www.rabbitmq.com/tutorials/amqp-concepts.html>. [Accessed: 14-Jul-2014].
- [53] Pivotal Software, Inc., “RabbitMQ - RabbitMQ tutorial - ‘Hello World!’,” *RabbitMQ*, 2014. [Online]. Available: <https://www.rabbitmq.com/tutorials/tutorial-one-java.html>. [Accessed: 14-Jul-2014].
- [54] Pivotal Software, Inc., “RabbitMQ - Clustering Guide,” *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/clustering.html>. [Accessed: 15-Jul-2014].
- [55] Pivotal Software, Inc., “RabbitMQ - Highly Available Queues,” *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/ha.html>. [Accessed: 22-Jul-2014].

- [56] Pivotal Software, Inc., “RabbitMQ - Distributed RabbitMQ brokers,” *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/distributed.html>. [Accessed: 15-Jul-2014].
- [57] Pivotal Software, Inc., “RabbitMQ - Management Plugin,” *RabbitMQ*, 2014. [Online]. Available: <https://www.rabbitmq.com/management.html>. [Accessed: 14-Jul-2014].
- [58] Jun Rao, “Clients - Apache Kafka - Apache Software Foundation,” *Apache Kafka wiki*, 02-Jul-2014. [Online]. Available: <https://cwiki.apache.org/confluence/display/KAFKA/Clients>. [Accessed: 15-Jul-2014].
- [59] BSON, “BSON - Binary JSON,” *BSON*, 2014. [Online]. Available: <http://bsonspec.org/>. [Accessed: 16-Jul-2014].
- [60] The Apache Software Foundation, “Apache Avro™ 1.7.6 Documentation,” *Apache Avro*, 23-Jan-2014. [Online]. Available: <http://avro.apache.org/docs/current/>. [Accessed: 16-Jul-2014].
- [61] K. Maeda, “Performance evaluation of object serialization libraries in XML, JSON and binary formats,” in *Digital Information and Communication Technology and its Applications (DICTAP), 2012 Second International Conference on*, 2012, pp. 177–182.
- [62] Chris Curtin and Neha Narkhede, “Consumer Group Example - Apache Kafka - Apache Software Foundation,” *Apache Kafka wiki*, 11-Sep-2013. [Online]. Available: <https://cwiki.apache.org/confluence/display/KAFKA/Consumer+Group+Example>. [Accessed: 16-Jul-2014].
- [63] Nicolas Niclausse, “Tsung,” *Tsung*, 05-Feb-2013. [Online]. Available: <http://tsung.erlang-projects.org/>. [Accessed: 01-Jul-2014].
- [64] The Apache Software Foundation, “Apache JMeter - Apache JMeter™,” *Apache JMeter*, 05-Jan-2014. [Online]. Available: <http://jmeter.apache.org/>. [Accessed: 16-Jul-2014].
- [65] HP Research lab, “Welcome to the httpperf homepage,” *httpperf tool (HP labs)*, 30-Jan-2008. [Online]. Available: <http://www.hpl.hp.com/research/linux/httpperf/>. [Accessed: 25-Jun-2014].
- [66] Amazon Web Services, “AWS | Amazon EC2 | Instance Types,” *Amazon Web Services, Inc.*, 2014. [Online]. Available: <http://aws.amazon.com/ec2/instance-types/>. [Accessed: 16-Jul-2014].
- [67] Oracle, “The Java HotSpot Performance Engine Architecture,” *Oracle Whitepapers*, 19-Jun-2013. [Online]. Available: <http://www.oracle.com/technetwork/java/whitepaper-135217.html#3>. [Accessed: 16-Jul-2014].
- [68] Oracle, “Java Management Extensions (JMX),” *Oracle - Java SE Technologies*, 18-Feb-2008. [Online]. Available: <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>. [Accessed: 16-Jul-2014].
- [69] Jiri Sedlacek and Tomas Hurka, “Home — Project Kenai,” *VisualVM*, 01-Jul-2014. [Online]. Available: <http://visualvm.java.net/>. [Accessed: 16-Jul-2014].
- [70] Wikipedia contributors, “sar (Unix) - Wikipedia, the free encyclopedia,” *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia., 30-Jun-2013.
- [71] C. J. Zarowski, *Numerical Integration and Differentiation*. Hoboken, NJ, USA: Hoboken, NJ, USA: John Wiley & Sons, Inc., 2004.
- [72] MongoDB Inc., “Aggregation Pipeline — MongoDB Manual 2.6.4,” <https://github.com/mongodb/docs/blob/master/source/core/aggregation-pipeline.txt>,

- 04-Aug-2014. [Online]. Available: <http://docs.mongodb.org/manual/core/aggregation-pipeline/>. [Accessed: 19-Aug-2014].
- [73] MongoDB Inc., “Map-Reduce — MongoDB Manual 2.6.4,” <https://github.com/mongodb/docs/blob/master/source/core/map-reduce.txt>, 04-Apr-2014. [Online]. Available: <http://docs.mongodb.org/manual/core/map-reduce/>. [Accessed: 19-Aug-2014].
- [74] ISO/IEC Information Technology Task Force, “ISO/IEC 9075-1:2011 - Information technology -- Database languages -- SQL -- Part 1: Framework (SQL/Framework),” *ISO*, 04-Feb-2013. [Online]. Available: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=53681. [Accessed: 19-Aug-2014].
- [75] Rick Osborne, “SQL to MongoDB example,” <http://rickosborne.org/download/SQL-to-MongoDB.pdf>, 06-Mar-2010. [Online]. Available: <http://docs.mongodb.org/manual/tutorial/map-reduce-examples/>. [Accessed: 19-Aug-2014].
- [76] Microsoft Technet, “Using Common Table Expressions,” *Microsoft Technet - Microsoft SQL Server documentation*, 2008. [Online]. Available: [http://technet.microsoft.com/en-us/library/ms190766\(v=sql.105\).aspx](http://technet.microsoft.com/en-us/library/ms190766(v=sql.105).aspx). [Accessed: 19-Aug-2014].
- [77] Amazon Web Services, “AWS | Amazon Redshift - Cloud Data Warehouse Solutions,” *Amazon Web Services, Inc.*, 2014. [Online]. Available: <http://aws.amazon.com/redshift/>. [Accessed: 20-Apr-2014].
- [78] Amazon Web Services, “Amazon Redshift and PostgreSQL - Amazon Redshift,” *Amazon Web Services, Inc.*, 01-Dec-2012. [Online]. Available: http://docs.aws.amazon.com/redshift/latest/dg/c_redshift-and-postgres-sql.html. [Accessed: 20-Aug-2014].
- [79] Amazon Web Services, “Columnar storage - Amazon Redshift,” *Amazon Web Services, Inc.*, 01-Dec-2012. [Online]. Available: http://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html. [Accessed: 20-Aug-2014].
- [80] J. Patti, “Moving product recommendations from Hadoop to Redshift saves us time and money | Monetate Engineering Blog,” *Monetate Engineering blog*, 18-Jun-2014.
- .
- [81] Sean O'Connor, “Speeding things up with Redshift,” *Bitly Engineering Blog*, 25-Apr-2013..
- [82] Patrick Wendell and U.C. Berkeley AMPLab, “Big Data Benchmark,” *Big Data Benchmark*, Feb-2014. [Online]. Available: <https://amplab.cs.berkeley.edu/benchmark/>. [Accessed: 20-Aug-2014].
- [83] Frederic Lardinois, “Google Launches BigQuery Streaming For Real-Time, Big-Data Analytics | TechCrunch,” *TechCrunch*, 25-Mar-2014. [Online]. Available: <http://techcrunch.com/2014/03/25/google-launches-bigquery-streaming-for-real-time-big-data-analytics/>. [Accessed: 20-Aug-2014].
- [84] Google Inc., “Google BigQuery — Google Developers,” *Google BigQuery*, 09-Jun-2014. [Online]. Available: <https://developers.google.com/bigquery/?hl=FR>. [Accessed: 20-Aug-2014].
- [85] Google Inc., “Pricing - Google BigQuery — Google Developers,” *Google BigQuery*, 09-Jun-2014. [Online]. Available: <https://developers.google.com/bigquery/pricing?hl=FR#free>. [Accessed: 20-Aug-2014].

- [86] Amazon Web Services, "Loading Data - Amazon Redshift," *Amazon Web Services, Inc.*, 01-Dec-2012. [Online]. Available: http://docs.aws.amazon.com/redshift/latest/dg/t_Loading_data.html. [Accessed: 20-Aug-2014].
- [87] J. Kreps, "Questioning the Lambda Architecture - O'Reilly Radar," *O'Reilly Radar*, 02-Jul-2014. [Online]. Available: <http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html>. [Accessed: 22-Aug-2014].
- [88] James Kinley, "The Lambda architecture: principles for architecting realtime Big Data systems," *James Kinley's blog*, Aug-2013. [Online]. Available: <http://jameskinley.tumblr.com/post/37398560534/the-lambda-architecture-principles-for-architecting>. [Accessed: 22-Aug-2014].
- [89] The Apache Software Foundation, "Welcome to Apache™ Hadoop@!," *Apache™ Hadoop@*, 2014. [Online]. Available: <http://hadoop.apache.org/>. [Accessed: 20-Apr-2014].
- [90] David DeMaagd, "Re: Fetch request with correlation id 1171437 from client ReplicaFetcherThread-0-1 on partition [meetme,0] failed due to Leader not local for partition," *Kafka-users mailing list archives*, 28-Jun-2013. [Online]. Available: http://mail-archives.apache.org/mod_mbox/kafka-users/201306.mbox/%3C20130628182502.GE27793@ddemaagd-ld.linkedin.biz%3E. [Accessed: 15-Jul-2014].
- [91] MongoDB Inc., "Read Preference — MongoDB Manual 2.6.3," *MongoDB Manual 2.6.3*, 01-May-2014. [Online]. Available: <http://docs.mongodb.org/manual/core/read-preference/>. [Accessed: 18-Jul-2014].
- [92] Event Store LLP, "Event Store," *Event Store*, 2014. [Online]. Available: <http://geteventstore.com/>. [Accessed: 22-Jul-2014].
- [93] D. Duggan, *Service-Oriented Architecture*. Hoboken, NJ, USA: Hoboken, NJ, USA: John Wiley & Sons, Inc., 2012.
- [94] M. Barai, *Service Oriented Architecture with Java*. Packt Publishing, 2008.
- [95] Abel Avram, "The Strengths and Weaknesses of Microservices," *InfoQ*, 28-May-2014..
- [96] S. K. Shin and G. L. Sanders, "Denormalization strategies for data retrieval from data warehouses," *Decis. Support Syst.*, vol. 42, no. 1, pp. 267–282, Oct. 2006.
- [97] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [98] MongoDB Inc., "What Is MongoDB?," *MongoDB*, Jan-2015. [Online]. Available: <http://www.mongodb.com/what-is-mongodb2>. [Accessed: 16-Jan-2015].
- [99] Jon Masamitsu, "Our Collectors (Jon Masamitsu's Weblog)," *John Madamitsu's Weblog*, 01-Feb-2008..
- [100] B. T. Kurotsuchi, "The Wonders of Java Object Serialization," *Crossroads*, vol. 4, no. 2, pp. 3–8, Nov. 1997.
- [101] Pivotal Software, Inc., "RabbitMQ - Consumer Prefetch," *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/consumer-prefetch.html>. [Accessed: 14-Jul-2014].
- [102] Cisco Systems, "Network Topology Icons - Doing Business With Cisco - Cisco Systems," *Cisco Sytems*, 27-Jan-2013. [Online]. Available: <http://www.cisco.com/web/about/ac50/ac47/2.html>. [Accessed: 10-Jul-2014].
- [103] Microsoft Corporation, "New Office Visio Stencil," *Microsoft Download Center*, 22-Apr-2014. [Online]. Available: <http://www.microsoft.com/en-us/download/details.aspx?id=35772>. [Accessed: 10-Jul-2014].

- [104] Michael G. Noll, "Running a Multi-Broker Apache Kafka 0.8 Cluster on a Single Node," *Michael G. Noll's blog*, 13-May-2013. .
- [105] GNU, "nohup invocation - GNU Coreutils," *GNU core utilities documentation*, Unknown. [Online]. Available: http://www.gnu.org/software/coreutils/manual/html_node/nohup-invocation.html. [Accessed: 14-Jul-2014].
- [106] Claude Mamo, "kafka-web-console - A web console for Apache Kafka," *GitHub*, 02-Jul-2014. [Online]. Available: <https://github.com/claudemamo/kafka-web-console>. [Accessed: 14-Jul-2014].
- [107] Pivotal Software, Inc., "RabbitMQ - Installing on RPM-based Linux (CentOS, Fedora, OpenSuse, RedHat)," *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/install-rpm.html>. [Accessed: 15-Jul-2014].
- [108] Pivotal Software, Inc., "RabbitMQ - Configuration," *RabbitMQ*, 2014. [Online]. Available: <http://www.rabbitmq.com/configure.html>. [Accessed: 15-Jul-2014].
- [109] Rémi Prévost, "Homebrew — The missing package manager for OS X," *Homebrew*, 2014. [Online]. Available: <http://brew.sh/>. [Accessed: 18-Jul-2014].
- [110] Takeharu Oshida, "Setup tsung for AmazonLinux," *GitHub*, 03-Apr-2014. [Online]. Available: <https://github.com/ngocdaothanhs/tsart>. [Accessed: 04-Jul-2014].
- [111] T1000, "tsung_stats.pl on Mac OS X Mavericks run into 'Can't locate Template.pm' error," *Stack Overflow*, 28-May-2014. [Online]. Available: <http://stackoverflow.com/questions/23912087/tsung-stats-pl-on-mac-os-x-mavericks-run-into-cant-locate-template-pm-error>. [Accessed: 21-Jul-2014].
- [112] bukzor, "ssh configuration: override the default username," *Stack Overflow*, 17-Apr-2012. [Online]. Available: <http://stackoverflow.com/questions/10197559/ssh-configuration-override-the-default-username>. [Accessed: 21-Jul-2014].
- [113] Tom Feiner, "Why does an SSH remote command get fewer environment variables then when run manually?," *Stack Overflow*, 19-Oct-2008. [Online]. Available: <http://stackoverflow.com/questions/216202/why-does-an-ssh-remote-command-get-fewer-environment-variables-then-when-run-man>. [Accessed: 02-Jul-2014].
- [114] Tom Maguire, "Erlang: Cannot start slave - {error,timeout}," *Stack Overflow*, 29-May-2012. [Online]. Available: <http://stackoverflow.com/questions/10796206/erlang-cannot-start-slave-error-timeout>. [Accessed: 21-Jul-2014].
- [115] Tsung, "10. Frequently Asked Questions — Tsung 1.5.1 documentation," *Tsung documentation*, 2013. [Online]. Available: http://tsung.erlang-projects.org/user_manual/faq.html#what-is-the-format-of-the-stats-file-tsung-log. [Accessed: 23-Jul-2014].

Appendix A — Apache Kafka – Quick reference

This section can serve as a quick reference for users and administrators that may want to try, install and manage a Kafka installation. For the sake of reproducibility of the tests presented in this document, the links and commands given here are specific to Apache Kafka 0.8.1.1. Some documents listed in the references may be handy as well [104].

The following commands have been tested successfully on Mac OS X 10.9.4 (13E28) and Red Hat 4.4.7-4 (Linux version 2.6.32-431.11.2.el6.x86_64). Though, installation instructions may vary from one system to another.

1. Install Kafka on the broker

```
# Install the dependencies (Scala)
sudo yum install scala
```

```
# Download the binaries
```

```
cd ~
```

```
wget
```

```
https://www.apache.org/dyn/closer.cgi?path=/kafka/0.8.1.1/kafka_2.9.2-0.8.1.1.tgz
```

```
tar -xvf kafka_2.9.2-0.8.1.1.tgz
```

```
# Move Kafka to its final location
```

```
mv kafka_2.9.2-0.8.1.1/ /opt/kafka
```

2. Edit the brokers' and Zookeeper preferences

You may find the preferences to set up the Zookeeper instance in `/opt/kafka/config/zookeeper.properties`. You can find the example configuration file for a Kafka broker in `opt /kafka/config/server.properties`. Duplicate this file if you want to run multiple logical brokers on the same machine (for testing purposes only).

3. Start Zookeeper and Kafka

One may find the `nohup` Linux command useful to run processes in background [105]:

```
# Zookeeper
```

```
nohup /opt/kafka/bin/zookeeper-server-start.sh
```

```
/opt/kafka/config/zookeeper.properties >
```

```
/opt/kafka/log/zookeeper.log 2>&1 &
```

```
# Kafka broker
```

```
nohup /opt/kafka/bin/kafka-server-start.sh
```

```
/opt/kafka/config/server.properties > /opt/kafka/log/broker.log 2>&1
```

```
&
```

Using these commands allow the processes to run in background. They will stay up and running even though the user disconnects from the machine (e.g. in case processes were launched through an SSH connection). Outputs are redirected to the log files `/opt/kafka/log/zookeeper.log` and `/opt/kafka/log/broker.log`.

4. Updated ZooKeeper CLI tool

The ZooKeeper management tool bundled with Kafka is an old version that among other things lacks the ability to recursively prune parts of the shared tree maintained in ZooKeeper, and therefore requires all the nodes' contents to be prior to the prune operation. This makes delete operations (e.g. removing a topic) very tedious tasks. It is strongly recommended to switch to an updated version of the ZooKeeper command line to perform those operations with more comfort.

```
# Get an updated Zookeeper CLI (that has the convenient rmr function
to recursively remove branches)
```

```
http://apache.mirrors.spacedump.net/zookeeper/stable/
```

```
tar -xvf [file]
```

```
# Access the ZooKeeper instance using the updated CLI
cd zookeeper-[version]/ && bin/zkCli.sh
```

5. Management operations

Here follows some of the most common set up and management operations one may need to perform on a Kafka broker (the ZooKeeper instance is assumed to be run on the same machine as the one that commands are executed from, with defaults settings):

```
# List the topics of a Zookeeper instance
cd /opt/kafka/ && bin/kafka-topics.sh --list --zookeeper
localhost:2181
```

```
# List consumers of a given consumer group, with their states
cd /opt/kafka/ && bin/kafka-run-class.sh
kafka.tools.ConsumerOffsetChecker --zkconnect localhost:2181 --group
[GROUP NAME]
```

```
# Create a topic
cd /opt/kafka/ && bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor [REPLICATION FACTOR] --
partitions [NUMBER OF PARTITIONS] --topic [TOPIC NAME]
```

```
# Start to consume raw data of a topic (for testing purpose)
cd /opt/kafka/ && bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic [TOPIC NAME]
```

```
# Read the content of a topic from the beginning (for testing
purpose)
cd /opt/kafka/ && bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic [TOPIC NAME] --from-beginning

# Delete a topic ----
# Access the ZooKeeper instance using the updated CLI
zookeeper-3.4.6/ && bin/zkCli.sh

# Delete the following metadata in Zookeeper
rmr /brokers/topics/TOPIC-NAME
rmr /consumers/[groupId]/owners/TOPIC-NAME
rmr /consumers/[groupId]/offsets/TOPIC-NAME

# Delete the topic queue data from the disk
rm -R /tmp/kafka-logs/[TOPIC-NAME]
# -----
```

6. A Web UI for Kafka: kafka-web-console

Apache Kafka lacks a proper administration interface. Some independent developers tried though to create a web interface to make it easier to watch and perform some operations on Kafka architectures. One of the most famous is kafka-web-console [106].

```
# Install kafka-web-console
# sbt is needed (available in brew for Mac)
# Clone the git repository
git clone https://github.com/claudemamo/kafka-web-console.git

# Put Play in the kafka-web-console directory
http://www.playframework.com/download

# Add the following line in conf/application.conf file
applyEvolutions.default=true

# Run activator
cd kafka-web-console/ && ./activator
(...)
[kafka-web-console] $ start

# Access the console
http://localhost:9000
```


Appendix B — RabbitMQ – Quick reference

This section contains all the information needed to get started with RabbitMQ 3.3.4, from installation to management. The reader may find additional resources in the References section 10 page 92.

The following commands have been tested successfully on Mac OS X 10.9.4 (13E28) and Red Hat 4.4.7-4 (Linux version 2.6.32-431.11.2.el6.x86_64).

1. Install RabbitMQ on the broker

On Mac OS X, one can directly download RabbitMQ from the RabbitMQ's website (<http://www.rabbitmq.com/install-standalone-mac.html>) with a bundled runtime version of Erlang.

On Red Hat, one need to perform some extra steps to be able to install RabbitMQ flawlessly [107]:

```
# Add the EPEL (Extra Packages for Enterprise Linux) repository,
which is an official Red Hat-maintained repository
rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-
release-6-8.noarch.rpm

# Install Erlang
yum install erlang

# Add the signing key of RabbitMQ
rpm --import http://www.rabbitmq.com/rabbitmq-signing-key-public.asc

# Download and install the package
wget http://www.rabbitmq.com/releases/rabbitmq-
server/v3.3.4/rabbitmq-server-3.3.4-1.noarch.rpm
yum install rabbitmq-server-3.3.4-1.noarch.rpm
```

2. Edit RabbitMQ preferences

Though probably unneeded, one may find (or create if it does not exist) the configuration file in `/etc/rabbitmq/rabbitmq.config`. Please refer to the documentation for more information about the available configuration variables [108].

3. Start and stop RabbitMQ

In most packaged versions, RabbitMQ is available as a service called `rabbitmq-server` and as such can be managed using the typical service `rabbitmq-server start/stop/...` command invocation.

On Mac OS X, it can be run using `sbin/rabbitmq-server`, or can be executed as daemon via `sbin/rabbitmq-server -detached`.

4. Enable and use the management UI

RabbitMQ comes bundled with some optional plugins. One very useful is the web management UI that can be simply activated with this command:

```
/sbin/rabbitmq-plugins enable rabbitmq_management
```

It may be required to restart the broker for the change to be effective. Once it is restarted, the management UI can be reached at <http://localhost:15672>. The default credentials are “guest” for both the username and password.

Please note that the default settings only allow connections from *localhost* to connect to the management interface using the default credentials. To connect producers and consumers to the broker, a specific account with a different password will also be needed.

5. Python script to parse queue logs

The *rabbitmq-management* plugin plots graphs in the web management interface. However, to make statistical analysis, extracting the values may be required. The following Python script may be used to convert the JSON structure that the RabbitMQ API can generate about a queue to a CSV file. The script also computes message rates, and adapts to different time interval.

You may get the JSON structure required as input for this script by issuing the following HTTP GET request:

```
http://[management_URL]/api/queues/%2F/[queue_name]?lengths_age=800&lengths_incr=5&msg_rates_age=800&msg_rates_incr=5
```

```
import json
import sys

# Script to process RBMQ statics and convert them to CSV
# Two arguments required:
# 1. input file (JSON structure)
# 2. output file (CSV file)

if(len(sys.argv) < 3):
    print "You need to pass two argument:\n1. input file (JSON structure)\n2. output file (CSV file)"
    sys.exit()

# Open the JSON file
f = open(sys.argv[1], 'r')

# Open the destination file
fOut = open(sys.argv[2], 'w')

# Load the JSON structure
obj = json.loads(f.readline())
f.close()

# obj keys : [u'auto_delete', u'deliveries', u'idle_since',
u'message_stats', u'messages_unacknowledged', u'consumers', u'durable',
```

```
u'state', u'arguments', u'memory', u'exclusive_consumer_tag',
u'messages_ready_details', u'consumer_utilisation', u'node',
u'messages_details', u'vhost', u'consumer_details',
u'messages_unacknowledged_details', u'incoming', u'name', u'policy',
u'backing_queue_status', u'messages', u'messages_ready']

numPt = len(obj['messages_details']['samples'])-1

title = "Timestamp [s], Total messages (queued+in process), Unacked
messages (in process), Queued messages, Publish rate [mes/s], Deliver rate
[mes/s], Acknowledgment rate [mes/s]\n"

interval =
(obj['incoming'][0]['stats']['publish_details']['samples'][0]['timestamp']
-
obj['incoming'][0]['stats']['publish_details']['samples'][1]['timestamp'])
/1000
time=numPt*interval

out = ""

# The file is reordered: the array is read from the beginning to the end
but output data is sorted in reverted order with the most recent times at
the end of the file
for i in range(0,numPt):
    time = time-interval
    line = str(time)+", "
    line+= str(obj['messages_details']['samples'][i]['sample'])+", "
    line+=
str(obj['messages_unacknowledged_details']['samples'][i]['sample'])+", "
    line+= str(obj['messages_ready_details']['samples'][i]['sample'])+", "
    line+=
str((obj['message_stats']['publish_details']['samples'][i]['sample']-
obj['message_stats']['publish_details']['samples'][i+1]['sample'])/interval)+", "
    line+=
str((obj['message_stats']['deliver_get_details']['samples'][i]['sample']-
obj['message_stats']['deliver_get_details']['samples'][i+1]['sample'])/interval)+", "
    line+=
str((obj['message_stats']['ack_details']['samples'][i]['sample']-
obj['message_stats']['ack_details']['samples'][i+1]['sample'])/interval)+"
\n"

    out = line + out

# Add a header
out=title+out

fOut.write(out)
fOut.close()
```

Appendix C — Tsung – Quick reference

This section gives interesting information for beginners with *Tsung* [63]. It covers only HTTP testing (while *Tsung* can stress test many other protocols) and contains a list of problems encountered during its use and the solutions found to solve them.

The following instructions have been successfully tested on computers running Mac OS X 10.7 and Mac OS X 10.9. They should be adaptable to work on any typical Linux machine. It assumes that *homebrew* is installed on the computer [109]. The reader is invited to read the *Tsung* documentation once before going through those synthetic instructions and browse through the documents in references [110].

1. Download and install Tsung

```
# Install tsung on Mac via Homebrew  
brew install tsung
```

Tsung is then available via the command *tsung*.

2. Launch a test

From the test controller machine, issue the following command:

```
tsung -f [path to XML test file] -l [path to log folder] start
```

A new folder with the date and time as name will be created in the log folder that is passed to the command and the logs and results of the test will be put there.

3. Generate test reports

Tsung is able to generate synthetic test reports in HTML, with graphs and tables. To generate it, issue the following command:

```
cd [path to the folder with the results];  
/usr/local/Cellar/tsung/1.5.1/lib/tsung/bin/tsung_stats.pl; open  
report.html;
```

This will generate the report and automatically open it. The report is generated in the same folder as the results.

4. Frequently asked questions

This section gathers most of the solutions that were used when a problem with *Tsung* were faced.

1) Where are the examples said to be packaged with *Tsung*?

When *Tsung* is installed via *homebrew* (on Mac OS X), the examples may be found in `/usr/local/Cellar/tsung/1.5.1/share/doc/tsung/examples`.

2) A DTD must be set in the XML files used to define a test scenario. Which DTD should be use?

When *Tsung* is installed via *homebrew* (on Mac OS X), a usable DTD may be found in `/usr/local/Cellar/tsung/1.5.1/share/tsung/tsung-1.0.dtd`.

3) When generating the statistics using *tsung_stats.pl*, the script hangs or triggers “division by zero” errors. How to solve this problem?

Tsung calculates statistics every 10 seconds. Therefore, make sure that the tests are longer than 10 seconds, and that they generate enough queries during this period.

4) A lot of “connection refused” errors are reported when load gets high.

The number of dynamic TCP ports the operating system can use to connect to services is limited. By default, on Mac OS X 10.9, they are picked in the range `{49152; 65535}`, which allows 16383 simultaneous connections to be established. This may be not enough for certain load, thus leading to errors. On Mac OS X, this limit can be read and increased without having to restart using these commands:

```
# Read the current bounds of the dynamic TCP port range
sysctl net.inet.ip.portrange.first net.inet.ip.portrange.last
```

```
# Change the first port of the range
sudo sysctl -w net.inet.ip.portrange.first=20000
```

Note that the change does not survive a reboot and should therefore be made again.

5) When attempting to generate reports, *tsung_stats.pl* yields an error. How to solve it?

A required dependency in Perl is likely to be missing [111]. To install it, issue the following command: `sudo cpan Template`. Other errors may be related to *gnuplot*. Keep in mind that this script is provided as is for convenience only, and is not the way to go to extract metrics. See question 15) for more details.

6) A certain number of “error_connect_emfile” errors appear while testing.

The number of files that can be opened simultaneously is probably too low. The limit must be increased so that the kernel allows more file descriptors to be opened simultaneously. To know the current value of the limit, issue the following command:

```
ulimit -n
```

However, *ulimit* cannot be used on Mac to change the limit. Instead, add the following line in */etc/launchd.conf* (create it if the file does not exist) and restart:

```
limit maxfiles 50000 50000
```

Note that the change is definitive.

7) In the XML test description file, there is a valid IP address in the attribute *host* of either a *client* or a *server* tag (e.g. `<client host="10.0.0.1"/>`). However, it does not work and *Tsung* does not connect to it, why?

Tsung requires a hostname in the *host* attributes. If there is no available hostname for the needed IP address, modify the */etc/hosts* file to add an entry in the local DNS cache. Even if there is a valid DNS name in a remote DNS server, it may be a good idea to duplicate the entry to the local DNS cache, to prevent an eventual high DNS query load from affecting the test results.

8) Now there is a valid hostname “abc.def.com”, but it still does not work, why?

Tsung together with Erlang seem to have problems with domain names containing dots when referring to *Erlang* nodes. Try adding a line for the */etc/hosts* file to redefine the required domain name with the target IP address with a simple hostname, without any dot.

9) What are the steps to follow to add and test the connection to slaves for doing distributed tests?

All the following worked between Mac OS X machines. It should work on other systems as well though:

- i. Install *Tsung* on all the machines.
- ii. SSH keys need to be exchanged between the machines. The controller should be able to connect without password prompt to all the machines. Copy the public key of the controller to the slaves.
- iii. Test the SSH connection between the controller and the slaves. Do it for all the slaves, as the key fingerprints of all the slaves need to be approved.
- iv. Close the SSH connection and try to launch it again with the command `ssh yourSlaveHostname erl`. It is important to specify a hostname and not the IP address here. See questions 7) for more details. Do not specify a username before the hostname.

- v. If the connection fails without going any further, check your connectivity via *ping* (if the slave answers ICMP messages). Do not rely on *host*, *nslookup* or *dig* to make sure that the local DNS cache is properly set, as those commands contact DNS servers directly and bypass the local *hosts* file.
- vi. If SSH prompts for a password, then it is likely that the username on the controller is different from the username on the slave. See question 10) to solve this issue.
- vii. If the SSH connection is successful but the *erl* invocation fails with “*erl: command not found*” or something similar, there is likely a path problem. See question 11) to solve this problem.
- viii. If the SSH connection is successful and *erl* prompt is displayed, disconnect from the slave.
- ix. Test the establishment of connection through *Erlang* on the controller via the command: `erl -rsh ssh -sname test@dns-name-controller -setcookie mycookie`. Change “dns-name-controller” to the local DNS name of the controller from which you are initiating the connection. This command does not do anything yet.
- x. When *erlang* prompts for a command, type `slave:start("dns-name-slave",bar,"-setcookie mycookie")`. (including the trailing dot). Change dns-slave-name to the local DNS name set for the slave you are connecting to.
- xi. If everything is working, it should return `{ok,bar@dns-name-slave}` after a while. If it returns `{error,timeout}` or something similar, checkout question 12).

10)The connection made by *Tsung* fails with “ssh_askpass: exec(/usr/libexec/ssh-askpass): No such file or directory” in the log file. How to solve it?

Two reasons may lead to this kind of errors. First, be sure to follow the steps mentioned in question 9) and to attempt to connect at least once to the slave outside *Tsung*. If you can successfully connect to the machine (i.e. you have acknowledged the slave's key), then the problem is likely a difference between the username on the controller and the slave. There is no documented way to tell *Erlang* via *Tsung* to connect to a specific username. A possible workaround is to tell *ssh* on the controller to use automatically use a specific username for a given hostname [112]. Edit (or create) the `~/.ssh/config` file and add two lines per slave:

```
Host server-hostname
User username-on-slave
```

Change the permissions on this file via `chmod 600 ~/.ssh/config`. Remember to use as server hostname the hostname that is in `/etc/hosts`, and not the real fully qualified domain name of the server. Otherwise, *ssh* will not use the correct username. See question 8) for more information.

11) Despite the fact that *Erlang* is installed and works when invoking it through an interactive SSH session, slaves return “*erl: command not found*” when invoking *erl* directly in the *ssh* command. Why?

Non-interactive SSH sessions do not get the same PATH environment variable as interactive sessions [113]. Therefore, it is likely that the *erl* command could not be found in the paths specified in the PATH variable. To solve this issue, copy the PATH variable value of an interactive session (`echo $PATH`) and copy it to the file (create it if it does not exist) `~/.ssh/environment`. Do not forget to add the prefix `PATH=` at the beginning of the line.

A setting must also be changed in the *sshd* configuration file (`/etc/sshd_config`). Set to *yes* the setting `PermitUserEnvironment`. These changes must be performed on all slaves.

12) The SSH connection between the nodes is successful, a non-interactive *Erlang* invocation works too but the establishment of a connection through *Erlang* toward a remote node does not. Why?

Usually, these problems are DNS related. Check that the controller has a valid hostname set for the slave in its `/etc/hosts` file. Check also that the hostname of the computer is properly set. On Mac OS X, be sure to issue the command `hostname controller/slave-hostname` on your nodes to define the hostname by substituting the argument with the chosen hostname. Note that the change is not persistent and that the hostname may be reinitialised when the network connection is interrupted. After sleep mode, the hostname is likely to be reset. The current hostname can be checked by invoking the `hostname` command without arguments.

It is also required for slaves to have a valid DNS entry in their cache toward the controller. Make sure to add a line in their `/etc/hosts` file for the controller's hostname to its IP address. If the IP addresses change, do not forget to update them in all DNS caches.

13) *Erlang* does not want to connect to the slaves. How to make its output more verbose?

The logs about the SSH connection initiated by *Tsung* can be gathered using a small *ssh* wrapper shell script [114]. Create a *myssh.sh* file somewhere with this content:

```
#!/bin/sh

echo "$@" "$@" > /tmp/my-ssh.log
ssh -v "$@" 2>&1 | tee -a /tmp/my-ssh.log
```

Then, invoke *Erlang* from the controller with the following command (from the directory in which the shell script has been saved): `erl -rsh myssh.sh -sname test@dns-name-controller -setcookie mycookie`. In the *erlang* prompt, issue `slave:start("dns-name-slave",bar,"-setcookie mycookie")`. (including the trailing dot). Logs about the SSH connection between the nodes will be saved in `/tmp/my-ssh.log`.

14) The connection via *Erlang* to all the slave nodes is successful, but not toward *localhost* (i.e. the controller) when running a test, why?

The public key of the controller should be an approved key to connect to the controller itself as well, as *Tsung* seems to establish a SSH connection to the controller itself (the controller may not be the local computer). Once it is done, try to connect via *ssh* to the controller itself, and to invoke *erl* through a non-interactive console (see question 11)). If it works, try to launch the test again.

15) How can one get access to raw metrics after a test?

As a test is going on, a *tsung.log* file in the specified log folder is created that contains all the information about the test. New values are printed every 10 seconds. Please refer to the documentation to understand what the figures presented in this file mean [115].

5. Example Tsung XML test scenario description

Here follows the XML file that describes the test scenario presented in section 5.6 page 53. It may be used as an example for other test scenarios.

```
<?xml version="1.0"?>
<!DOCTYPE tsung SYSTEM "/usr/local/Cellar/tsung/1.5.1/share/tsung/tsung-1.0.dtd">
<tsung loglevel="notice" version="1.0">

  <clients>
    <client host="localhost" maxusers='30000' cpu="2" weight="2"/>
    <client host="nicolas-mba" maxusers='30000' cpu="2" weight="1">
      <ip value="192.168.0.249"></ip>
    </client>
  </clients>

  <servers>
    <server host="123.123.123.123.us-west-1.compute.amazonaws.com" port="80"
    type="tcp"></server>
  </servers>

  <load duration="290" unit="second">
    <arrivalphase phase="1" duration="30" unit="second">
      <users maxnumber="200000" arrivalrate="100" unit="second"></users>
    </arrivalphase>
    <arrivalphase phase="2" duration="30" unit="second">
      <users maxnumber="200000" arrivalrate="300" unit="second"></users>
    </arrivalphase>
    <arrivalphase phase="3" duration="30" unit="second">
      <users maxnumber="200000" arrivalrate="500" unit="second"></users>
    </arrivalphase>
    <arrivalphase phase="4" duration="30" unit="second">
      <users maxnumber="200000" arrivalrate="800" unit="second"></users>
    </arrivalphase>
    <arrivalphase phase="5" duration="40" unit="second">
      <users maxnumber="200000" arrivalrate="900" unit="second"></users>
    </arrivalphase>
    <arrivalphase phase="6" duration="40" unit="second">
      <users maxnumber="200000" arrivalrate="1000" unit="second"></users>
    </arrivalphase>
  </load>
</tsung>
```

```
</arrivalphase>
<arrivalphase phase="7" duration="40" unit="second">
  <users maxnumber="200000" arrivalrate="1300" unit="second"></users>
</arrivalphase>
<arrivalphase phase="8" duration="50" unit="second">
  <users maxnumber="300000" arrivalrate="1500" unit="second"></users>
</arrivalphase>
</load>

<options>
  <option type="ts_http" name="user_agent">
    <user_agent probability="100">Tsung load tester</user_agent>
  </option>
</options>

<sessions>
  <session name="http-example" probability="100" type="ts_http">
    <request>
      <http url="/event/log?sessionKey=3ae3dab2bb64c10f69ae99f4ed92818217a0ec67"
method="POST" version="1.1" contents='{ "eventType": "QUIT", "retentionTime": 100 }'>
      <http_header name="Content-Type" value="application/json; charset=UTF-
8"/>
    </http>
  </request>

  </session>
</sessions>
</tsung>
```

6. Helper scripts (Bash) to install Tsung, and deploy and run tests

Installing Tsung on Mac OS X and CentOS machine takes time, and it is possible to make mistakes at multiple steps. In order to ease the process of installing Tsung, and then running tests, either locally or remotely, multiple Bash scripts are available to download on the author's Bitbucket repository. These scripts are provided as-is without any warranty, but may really help one to deploy Tsung without too much hassle.

The scripts may be downloaded here: <https://bitbucket.org/nicolasnannoni/tsung-bash-helpers>.

TRITA TRITA-ICT-EX-2015:12