Abstract

Distributed systems rely on a way of interservice communication which represents the backbone of a system, as it is built around it. A Distributed system is built of several layers. At the top, the entry point is represented by the UI, typical portals that offer features to a customer, any action executed by the user goes to the infrastructure layer where the actions translate into events or requests that are routed to the software layer. The bigger a distributed system is, the more complicated the infrastructure layer is. In modern system it is comprised of application gateways, service meshes, internal networks with load balancers inside or Kubernetes clusters running services in containers that run the software layer. Once the software layer is finally hit, the actual transactions are executed and data is stored at the bottom, in the persistence layer. The infrastructure layer of a distributed system holds the utmost importance as it is required to safely transfer and route data over many nodes in the network graph. Losing data over the network, receiving it corrupted, or not receiving it at all can cause real damage inside the system. Depending on the system's needs, the communication could be done through https calls, gRPC, event streams or data streams. Over the years, each of these methods evolved into robust and reliable ways of putting a system together. One thing remained the same, a developer still has to handle various edge cases where unexpected things happen, dependencies may fail, or actually, the caller itself may experience issues and trigger a request several times. Solutions and design patterns exist to handle all the aforementioned issues, but this does not change the fact that one has to sometimes write more boilerplate code than actual business code, just to handle the 1% of situations where a fatal exception may occur. A distributed system has dependencies but it is also the dependency of its customer, and a dependency can never be trusted. It is considered best practice to retry a dependency that fails, or returns an unexpected result. While backoff retries are one of the best ways to ensure reliability, they can also cause the worst issues, as you can end up in a state where data is duplicated or corrupt. This thesis represents SafeBunny, a library based on communicating through messages, or events, built over RabbitMQ. It aims to remove all of the cases where a processing or network error would result in an irreversible transaction, that could derive the system into an invalid state at any point in time. SafeBunny tackles the issues of retries, corrupted messages, duplication and idempotence at a saga level, while keeping performance and extensibility in mind, with the amount of boilerplate code to a minimum. The topology of the broker is created behind the scenes, messages are automatically correlated between logical nodes and the routing is based on the ubiquity of the domain. All of these take away the labor of understanding the depths of a message broker, leaving the developer to follow the simple and well known Tell-Don't-Ask principle.

Contents

Abstract 1
1. Introduction 4
2. State of Art 6
  2.1 NServiceBus 6
  2.2 Convey 7
  2.3 Mass Transit 8
3. Technologies 9
  3.1 RabbitMQ.NET 9
  3.2 Polly 9
  3.3 CosmosDB 10
  3.4 Benchmark.NET 11
4. Wrapping RabbitMQ .NET 12
  4.1 Connection 12
  4.2 Channel 12
  4.3 Consumer 13
  4.4 Message Properties 14
  4.5 Delayed Retry 14
5. Architecture 15
  5.1 Pipeline 15
  5.2 Message Context 17
  5.3 Middlewares 17
  5.4 Circuit Breakers 18
  5.5 Subscription pipeline 19
  5.6 Publication pipeline 21
  5.7 Routing topology 22
6. Performance considerations 24
  6.1 Thread safety dangers 24
  6.2 Hot paths 26
    6.2.1 Reflection 26
    6.2.2 Fast Serialization 27
    6.2.3 Retries and RabbitMQ prefetch exhaustion 29

7. Deduplication 31
7.1 Deterministic Message Ids 32
7.2 CosmosDB as the transaction store 33
8. Performance 34
8.1 Publishing 34
8.2 Consuming 35
9. Conclusions 37
10. Future work 38
Bibliography 39

1. Introduction

Distributed systems face nondeterminism at every corner, beyond the scope of the current processing service, any external request is considered a dependency and any dependency can fail. Assuming that the network is deterministic for a second, another problem of todays distributed systems is multi-threading, but we don’t refer to a local system multi-threading, we refer to horizontal scaling. Here, depending on the sequence of requests issued by the consumer of a service, in the case of several instances of that service, any request can be handled by any instance of the service. Depending on the business logic, this can either be harmless or harmful. Consider two commands sent over the network, both update commands. Depending on the sequence they are executed in, we will expect different results, essentially pointing to us that our service as a whole, is not idempotent. For https calls this issue has been fixed by creating sticky load balancers, where one unique consumer will be served by one unique instance of the service, and the load is distributed instead by splitting users in subsets, each getting specific instances. If we move to event streaming, the topic of this thesis, such a solution is not feasible, since the context of the problem is completely different.

Amazon classifies distributed systems in three types [1]. *Offline* distributed systems, where race conditions do not matter, and anything can be safely retried until completion. *Soft real-time* systems, where systems have to eventually answer but the time-window to do that is a lot more lenient. This means that a distributed system can configure its retries mechanism in such a way that it will ultimately converge to a valid state. The last type is the *Hard real-time* system, where the consumer expects an immediate answer, that can be used for further transactions. Having an error or invalid state happen inside a real-time system, can cause hidden errors that will spread exponentially to other parts of the system.

Because of the nondeterministic nature of message consumption, we cannot know which instance will process a message, as they are usually served in a round robin manner. This leaves the system to handle unique race conditions or invalid states. Idempotency tokens, timestamps or acid databases are techniques employed to handle race conditions between instances. Distributed systems can theoretically scale to infinity, however, at high throughput, these dependencies become the bottleneck of the system, reaching maximum throughput before their CAP properties break. Once a dependency breaks, either because the maximum throughput has been reached, or something unexpected happened, the service has the option to either rollback, and let the consumer try again or retry the transaction. Here, message delivery systems split in two categories, at-most-once delivery and at-least-once delivery. One guaranteeing what the other cannot. If a dependency fails, you can safely rollback and let the end consumer try again, or safely try again but expect race-conditions. *Microsoft Orleans* [2] is an example of at-most-once delivery.

This thesis focuses on at-least-once delivery but ensures that critical transactions happen exactly-once behind the scenes. Exactly-once, as proposed by contributors of the NServiceBus framework, suggests that a message may be processed several times, but the side effects of that action can be deduplicated [3].

Beyond the scope of critical transactions, another important thing is ease-of-development. Understanding a message broker can be hard, and correctly using it is even harder. Ease-of-development translates in faster time-to-market, and in the real world of business, this is almost as equally important as having correctly functioning software. Often times, a business will accept various software bugs if it means that a product can be launched faster. With that in mind, this thesis proposes a modular architecture where the developer can essentially plug in anything, and several ways in which the message broker can be abstracted on a deeper layer, enabling the developer to focus on business logic.

2. State of Art

Multiple solutions exist for message-based communications solutions. Most of them are free to use and users can pay for support, or users have to pay a subscription depending on the scale of their application. Distributed libraries that support message-based communication do so by focusing their login on the communication part and abstracting the broker behind the scenes. By doing so they are able to support several types of brokers, such as RabbitMQ, ActiveMQ, SQL databases even or Azure Storage Queues/Storage Bus and so on, leaving the user to choose and the library behaves the same. SafeBunny on the other hand, focuses solely on RabbitMQ.

The reason for which multiple libraries appear is that each of them focuses on the different things, that were important in the eyes of the creator. Depending on the application that uses them, each have advantages and disadvantages.

For the most part, all libraries do the *RabbitMQ Essentials*, meaning they have separate connections for consuming and publishing messages, they implement a system for handling ghost messages or transactional errors, they have a set routing topology, or multiple from the users can choose, all the queues and exchanges are durable so messages are never lost and a poison queue serves as the last resort of the system.

2.1 NServiceBus

NServiceBus is a subscription-based message-based communication library. Beyond the message library part, their package also comes with a nice web application which serves as a live monitor regardless of the message broker. They bring information regarding message rates, errors, concurrency errors and health states of logical endpoints. For the State of Art, we will be analyzing their RabbitMQ implementation inside NServiceBus.

A beautiful feature of NServiceBus is that they support multiple topologies, from which the user can choose. This enables applications to pick the best topology for the highest throughput and performance in their context.

In the case of delayed message delivery, NServiceBus also uses the delayed exchange plugin provided by the RabbitMQ team. What NServiceBus does different, is that they start a connection per channel, unlike SafeBunny where channels are multiplexed over one connection regarding that service. NServiceBus does this out of principle, as an architecture design where they enforce logical endpoints to consume one type of message, meaning a process using NServiceBus is only able to consume a single queue. While being a good thing, it’s also not the best for smaller applications. This approach shines in large applications where individual service throughput is harder to achieve. It is also cheaper to have multiple subscribers in the same process for smaller applications.

They also offer Administration Tools, that can be used for broker management, creating queues and exchanges or deleting them. NServiceBus also approaches graceful shutdowns, where in case if disposal, the library waits for all services to finish their ongoing tasks and messages to be routed or consumed, before triggering the final disposal. The same process is also applied for their circuit breaker.

An optional service, “Inbox/Outbox” is offered, which can be activated by the user. It serves as a plugin which helps with avoiding ghost messages, and sending messages *exactly-once*. It uses a persistence store, either SQL or NoSQL in order to save messages states and relevant metadata to achieve durable states across instances. The only present downside of this plugin is that the user has to know how to use it, he must be able to know what should be placed in the outbox bucket and why it should be there, the system cannot be used transparently. When used correctly, Inbox/Outbox proved to have great results.

In order to handle the concurrency issues of publishing, NServiceBus resorts to a channel container with a Concurrent Queue behind it, each channel takings its turn in order to publish all messages of a scope to the message broker. If there are not enough channels NServiceBus simply creates more with no apparent upper limit.

NServiceBus is used in many production instances and proved to be a reliable message-based communication library with many uses in many different contexts.

2.2 Convey

Convey is actually a .NET library stack for microservices, it has everything from APIs, Middlewares, Authentication, CQRS integrations, service discovery and balancing, distributed tracing and a small library for interacting with RabbitMQ which is designed to integrate perfectly with the Convey eco system.

Convey implements a lighter version of the Inbox/Outbox system as NServiceBus, supporting both Entity Framework or Mongo, and unlike NServiceBus, Convey multiplexes channels over one connection in the case of message consumption.

For publishing, Convey makes use of a channel pool as well, however their approach is different, they rely on the managed thread Id in .NET as a key for a channel, with a max channel number, and a new channel is request for every publish to the broker. In later version of .NET, the managed thread Id is no longer the same as the OS, meaning that .NET can spawn multiple more threads than the OS allows and Convey could be left with unused channels between throughput spikes. Besides this channel leak, Convey throws an exception if the channel limit is hit, instead of waiting for a channel to be released, which translates into wasted message retries and potential duplicated transactions.

2.3 Mass Transit

Mass Transit is one of the most popular open-sourced RabbitMQ libraries for .NET Core. Mass Transits implementation of RabbitMQ is one of the biggest and most thorough implementation, covering pretty much everything that an application needs from a message broker a lot more different feature. Depending on the application, Mass Transit is not the lightest but is the most robust.

One thing that sets apart Mass Transit from other libraries, is the routing topology and the message contracts. Mass Transit enforces assembly matching message consumption and publication, because the routing topology is built on the namespace of the types. The drawback to this is explained in [Ch 5.7]. If the namespace is not respected, messages are lost, and Mass Transit does not offer an alternative to this topology.

Similar to Convey and unlike NServiceBus, Mass Transit allows multiple subscriptions from the same process and most of its configuration is code first, which can be a little challenging at first, however Mass Transits documentation is very well done.

In the case of concurrency and protection from ghost messages or duplicated transactions, Mass Transit does not offer Inbox or Outbox and it actually offers a Saga mechanism, which developers can use. Any operation done in a Saga is persisted in a SQL or NoSQL database, many of which are supported. Mass Transits persists the state of the saga and relevant metadata regarding previous operations and the state of success. This functionality is not provided outside sagas.

For delayed messaging, Mass Transits does not use the delayed messaging plugin of RabbitMQ, and puts the stress of delaying on the instance that wants to publish the message. It implements Quartz .NET for scheduled messaging, which stores future jobs, in this case messages, in a database and publishes them at the scheduled time. This comes as an advantage in the case of message broker shutdowns, because delayed messages are stored in RabbitMQ’s RAM and they are not persisted upon restarts.

Mass Transit is also the first well known message bus library to adopt CosmosDB as a persistence store the state of messages, however it is only used in sagas.

3. Technologies

Several technologies were used in order to create SafeBunny, each chosen for a specific purpose. In order for the library to be as maintainable as possible, the number of external dependencies need to be kept minimal. Technologies range from the language the library was built in, SaaS services, and important plugins.

SafeBunny is built as a .NET Standard package. This means that it can be used in the entire ASP.NET ecosystem, as it compatible with both .NET Core and .NET Framework. This makes the range of possible customers as high as it gets. Through development, SafeBunny was under a CI\CD pipeline with SonarQube built into it. This ensured that code quality was kept to high standards.

3.1 RabbitMQ.NET

RabbitMQ is the message broker that SafeBunny relies on. It has an official .NET Standard library which contains the minimum code in order to create a connection to the broker, send and receive messages from it. SafeBunny builds over it in order to offer more functionality. RabbitMQ.NET [4] is an implementation of the AMQP 0.9.1 protocol specific for RabbitMQ. Even though it is built specifically for .NET, there are several drawbacks and dangers that come with using this library, such as possible memory leaks, thread safety, and communication between threads. These are explained in [Ch 6.1] SafeBunny resolves all of them in an easier manner for the end-user. The RabbitMQ.NET library offers support for administrating queues and bindings, receiving confirmations on any action and topology recovery in case of a disconnection. SafeBunny builds over the topology recovery with a specific one that is used in the circuit breaker [Ch 5.4].

3.2 Polly

Polly [5] is a .NET Library that turns otherwise vulnerable executions into resilient ones. It offers features such as Back Off Retries; Jitter reties and circuit breaker. The decision to use Polly was due the fact that SafeBunny needed a way to safely retry in memory various types of calls, such as connecting to the message broker and executing user code. The other option was to manually implement such a mechanism however that would involve being careful about threads, .NET tasks optimizations, control of delays and configurable timeouts. Since Polly is proven in many production workloads and well known for offering resilient code to solve exactly the needs of SafeBunny, the decision was taken to use Polly as the main mechanism for retries.

3.3 CosmosDB

SafeBunny needs a way to store certain data during the deduplication process. The requirements of this process are high speed, high throughput and the data needs to be instantly replicated and accessible across any point that would be required in the deduplication context. In addition to that, the client that connects to the database must be lightweight in order not to interference with the applications performance.

Typical acid databases do not satisfy these needs at high throughput, as several transactions, even though they are mutually exclusive, would not be processed in parallel depending on the architecture of the tables. In the case of database replication, read-last-write is essential for SafeBunny, and in the context of a message bus where the throughput of messages is basically constant, the latency pushed by replication inside a cluster can become the real bottleneck in performance.

In the case of NoSQL databases, the constraints are less however they are still there. First things first, NoSQL does not support unique keys by design. This can be circumvented in several ways however the last issue is the sharding process in a typical NoSQL database. In order for the maximum throughput of read-last-write to be achieved, SafeBunny would ideally read from specific shards of the database where it knows that for the current operation, that is the only place where data is found. As the number of transactions grows, one of two things happen, the shards are getting too big individually and data is lost, or the number of shards is getting too big and the data is lost

After clearing out the requirements, the most promising solution is CosmosDB [6], a NoSQL database, created by Microsoft which brings a few extra properties to NoSQL databases. The data is stored in containers, similar to collections in other databases, however in the case of CosmosDB, each container has its own throughput that is not affected by the load of other containers in the same database. The database can also be configured in shared throughput, but in the case of SafeBunny, maximum throughput is desirable. A container inside a CosmosDB database, is partitioned based on specific keys set at initialization. Based on these partition keys, the container will split in as many logical partitions as values for the partition key. A logical partition represents the scope of a transaction in CosmosDB and it also is the scope of reads and writes [7]. Compared to shards in other databases, this drastically improves performance, latency and lowers the problem space of a transaction in the case of SafeBunny.

CosmosDB ensures that for a specific logical partition the client should receive the latest write in the strong consistency level and at least its own writes and reads in a session consistency level. In addition, In the case of a message broker, full consistency is not required [Ch 7.2]. It is only necessary that duplicated writes are never permitted, which holds true in the scope of a logical partition. Because of how CosmosDB handles partitions and transactions, it is the best option for SafeBunny.

3.4 Benchmark.NET

Since SafeBunny is a message-based communication library and this means that almost the entire library is considered a hot-path, which can affect an application's performance. Because of this all of this, all code execution paths must be as performant as possible. Any decision that could affected performance has to be analyzed and motivated.

Benchmark .NET is used to measure the performance of all possible options in an architectural decision and choose the best. Memory allocation and execution times are the statistics of interest to SafeBunny, both offered by Benchmark .NET. The package offers different tools for ease of benchmarking, controlled environments, controlled .NET runtime, and controlled workloads. This means that before the actual benchmark, the package "warms-up" the CPU by running the workload multiple times until the mean stabilizes and then performance is calculated.

All benchmarks in this thesis are executed on the following configuration:

```
BenchmarkDotNet=v0.13.0
OS = Windows 10.0.17763.1935 (1809/October2018Update/Redstone5)
RAM = 64GB CL14 3000MHz
AMD Ryzen 7 2700X, 1 CPU, 16 logical and 8 physical cores
  [Host]     : .NET 5.0.4 (5.0.421.11614), X64 RyuJIT
  Job-GVEMID : .NET 5.0.4 (5.0.421.11614), X64 RyuJIT
Runtime=.NET 5.0
```

4. Wrapping RabbitMQ .NET

The RabbitMQ .NET library offers several primitives in order to establish a connection with the message broker, do Exchange/Queues management and several events that the consumer can hook to receive information about failures or confirmations.

4.1 Connection

SafeBunny builds over the default Connection with a *SafeBunnyConnectionFactory* which is responsible with creating a named connection. Regardless if the creation is successful or not, the connection factory will attempt to create the connection indefinitely using *Polly*. This behavior is desired, if a connection shuts down unexpectedly, event handlers will be triggered and immediately try to reconnect.

In order to separate concerns and to ensure as much thread safety as possible, SafeBunny opens one connection destined for *Consumption* and one connection for *Publishing*. The two actions are opposite, and so they happen on separate connections.

4.2 Channel

Channels are lightweight connections multiplexed over a connection. They are used in order to send and receive messages, commands or events from the message broker. SafeBunny uses channels in two different ways depending on their use case.

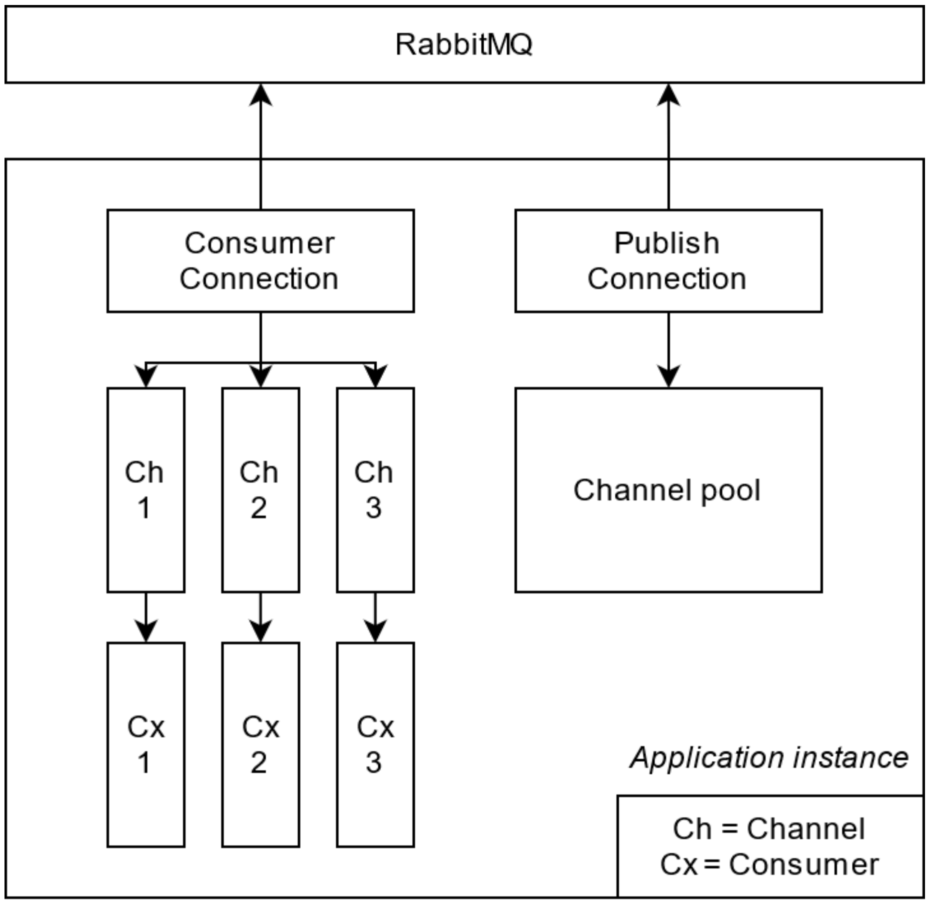
For message consumption, SafeBunny opens one channels destined to queue management, and it is closed once the application starts-up successfully. Once that is done, a channel is created per message type that the application desires to consume, since SafeBunny uses one channel for one queue.

In the case of publishing, the behavior is different. SafeBunny holds a pool of channels that are a one-publish-use. The reason for this is thread-safety dangers which are covered in [Ch 6.1]. The channels used in publishing are also wrapped around another class that holds extra publish-confirmation logic. Publishing in RabbitMQ is fire-and-forget by default, a message may or may not be delivered. Channels can be set *transactional*, meaning that a confirmation from the broker will arrive back as an event once a message is confirmed to have reached any queue. Because the event can arrive on another thread, the logic containing event handling is contained in the wrapper, together with the channel responsible of the publication.

4.3 Consumer

Consumers are used by channels to consume a specific queue. They can either be used with automatic ack or manual ack. SafeBunny uses manual-ack in order to tell the message broker if it decides to reject a message or confirm it as successfully consumed. Consumers also provide several events for successful registration, shutdown or unexpected shutdown. All three types of events are used in the Circuit Breaker [Ch 5.4], in order to control the connection to the message broker. Basic and Async consumers are available from the core library and SafeBunny only uses async consumers.

One consumer is allocated per active channel, translating further in one consumer per queue. All of them are held in memory for the lifetime of the application, recovered if lost and managed completely by the internals of SafeBunny. Each consumer can process a specific number of messages at once. This limitation can be set by the developer and it defaults to 10.



[Fig.4.1] Example of a connection to the broker with 3 subscriptions

4.4 Message Properties

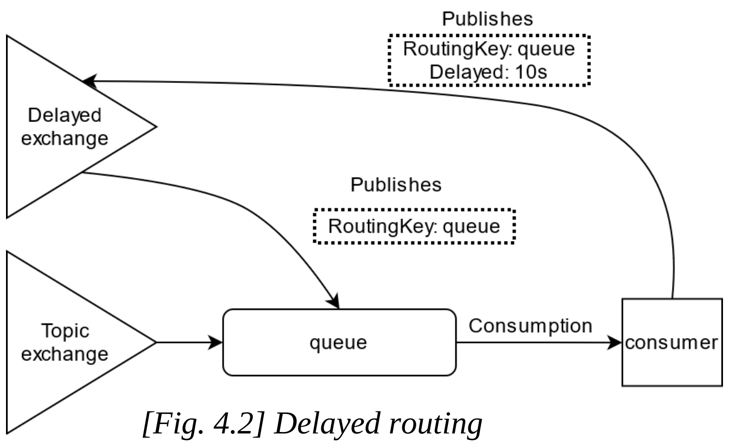
Every message that goes or comes from the message broker, has several properties attached to it. A message consists of the body, the actual data that someone wants to publish, and properties, metadata of the message that the broker needs in order to function. The default data consists of routing information, message information such as identifiers, encodings and failure state.

SafeBunny adds extra information to this in order to be able to offer more features in a distributed context, where services don't know of the existence of each other. The need for more information comes from features like resilience, retries, transactional publishing and client headers. SafeBunny adds extra data like correlation ids, message ids, status of retry attempts between processing, custom client headers which are accessible from the message context and unique SafeBunny identifiers that are used to confirm whether and which messages have successfully reached the message broker, or which are part of saga transaction.

4.5 Delayed Retry

Out of the box, RabbitMQ does not support sending messages in a delayed fashion, to a queue or to a consumer. Such a mechanism is needed and it is not feasible to hold a message in memory for *x* timespan, as it would hog memory and block other messages from being processed. A custom plugin [12] exists for RabbitMQ which enables exchanges to publish messages after a delayed amount of time. The plugin works by storing messages inside RabbitMQ's memory and delivering them to a specific queue after a determined amount of time.

SafeBunny integrates this plugin and adds a special exchange which uses this plugin. Any message that fails processing and is valid for message bus retries, is published to this exchange with a timer configured by the user and a routing key back to the queue where it was consumed from. The message is acknowledged and processed again.



[Fig. 4.2] Delayed routing

5. Architecture

In the context of message processing, where the entire library in considered a hotpath, it is necessary for the library to achieve the highest performance possible while being extendable and maintainable in the same time. Having this in mind, a modular architecture was chosen, similar to what Microsoft does for the request pipeline in a REST API [8], with several key differences. SafeBunny's architecture is built around processing any kind of action inside a generic pipeline, on processing contexts which are mutable only from inside library defined objects. This means that for a consumer to mutate the context, he would have to add a plugin to the pipeline, which can access and manipulate data.

Besides data processing, key parts of the library are the classes that initiate or hold the connection to the message broker. In the case of those, no particular type of architecture has been applied.

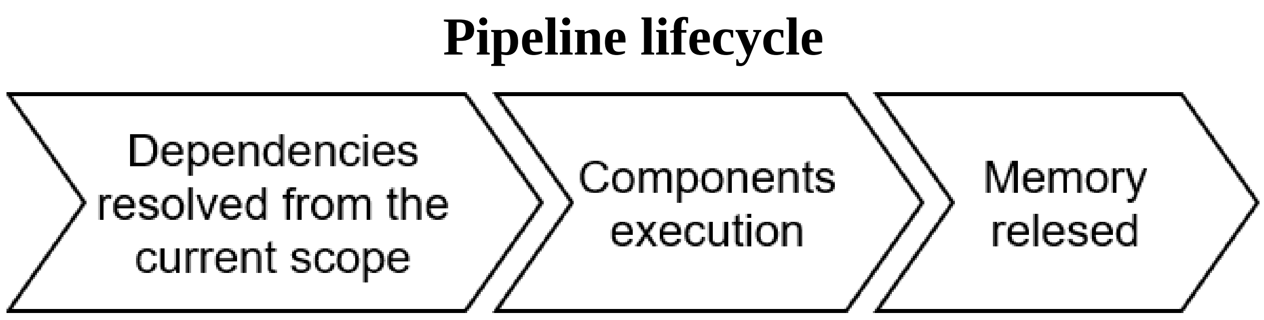
In the case of CosmosDB, the current logical node does not need several connections to CosmosDB, or connections to different databases. This means that for maximum performance, a single, long lasting connection to the database has been chosen.

5.1 Pipeline

Any kind of action that would involve communication to the message broker (*Message publishing*) or execution of a message handler (*Message processing*) is done by invoking a generic pipeline.

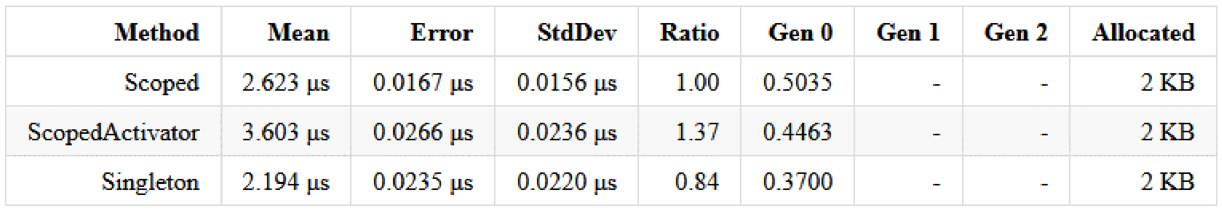
The entire pipeline is built around modular components, which we will call Middlewares. They are executed in order, in several execution groups. By execution group, it means that there are several points in the lifecycle of an action where a middleware could be injected. Before the action that the user wants to execute, and after the action has completed, regardless if it has been successful or not. Any middleware can short-circuit the pipeline if it decides that there is no value in any further processing. The execution of the action itself is abstracted in components that are part of SafeBunny, which cannot be changed or short circuited by the consumer.

As an input, the pipeline requires a context with a service scope, from which it will resolve the required services to execute the pipeline. If no service scope is present on the context, meaning that SafeBunny received a message and sent it to the pipeline, a new one will be resolved which will act as the scope for the rest of pipeline, or any secondary pipelines invoked during the pipeline execution. Secondary pipelines can exist, for example publishing a message off the current message that is being processed. This means that the subsequent pipelines will inherit the scope, and along it, the already resolved dependencies. By using this pattern, we ensure that any dependencies resolved will always hold the same reference for the current message that is being processed, and also be released when the message has finished processing.



[Fig. 5.1] For the context that initializes the pipeline, dependencies are resolved, pipeline is executed and then memory is released.

Resolving components and releasing them in the scope of every pipeline invocation, comes with advantages and disadvantages. The alternative to this, would be to have all middlewares act as singletons, together with the message handler implemented by the client. Greater performance would be achieved by not having to allocate memory to every single dependency, however, since the message handler is at the top of the dependency chain, the consumer dependencies would all be forced to be singleton as well. This implies all sorts of possible memory leaks so the decision was to allocate any necessary memory at the start of the pipeline, and then release it.



Method	Mean	Error	StdDev	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Scoped	2.623 μs	0.0167 μs	0.0156 μs	1.00	0.5035	-	-	2 KB
ScopedActivator	3.603 μs	0.0266 μs	0.0236 μs	1.37	0.4463	-	-	2 KB
Singleton	2.194 μs	0.0235 μs	0.0220 μs	0.84	0.3700	-	-	2 KB

[Fig 5.2] The scoped pipeline performs 37% better than manual component resolving and only 16% worse than a singleton approach.

This benchmark showcases a pipeline built of two core components and two simple middlewares, where scoped dependency activation serves as a baseline (*Ratio)*. There is no message handler in the benchmarked pipeline since any user code should not be part of the library performance considerations.

The last important decision was either if the middlewares should be part of the service descriptors in the running application, or created and destroyed by the pipeline. The (*Gen 0*) memory allocation is less however the performance is overshadowed by the default scoped behavior of .NET. Since execution time is more important and the required extra memory is low, the middlewares are also part of the application's service descriptors.

5.2 Message Context

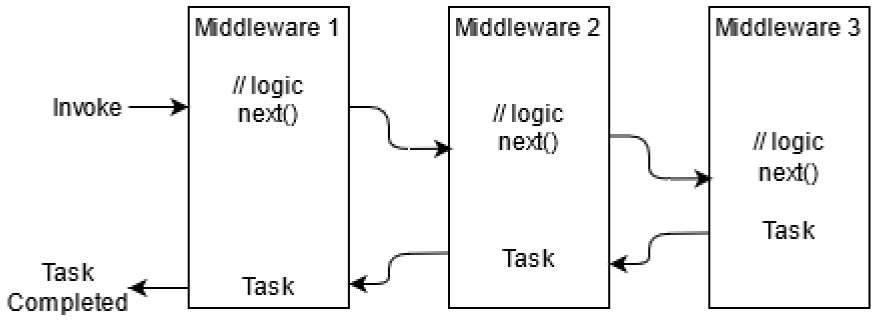
Contexts are the input of pipelines and what user actions revolve around. They serve all the information necessary to assess the state of a message that is currently being processed.

In the case of a message that is being consumed from the message broker, a context will contain all the metadata of that specific message type, reflection information, retry or failure status, the actual message content, message properties and the resolved scope. From a developer standpoint, only a few properties are visible, there is no real need for a developer to have access to scopes or reflection metadata. None of the properties are mutable, however they can be transformed to another message. The processing context is built once a message is received from the broker. If a handler exists for the message, the channel is still open and deserialization is successful, the message is considered valid and the pipeline is invoked.

For a publishing action, a pipeline is invoked using a publishing context containing just the message body and the properties of that message.

5.3 Middlewares

Middlewares are singular components which are linked together by the pipeline. The next middleware in order can be called through the next reference. All middlewares are implemented the same regardless of their desired injection place. That is decided at application startup by registering the middleware in a specific execution group.



[Fig. 5.3] Middleware execution order and inter-linking

The entire pipeline can be executed multiple times from a specific middleware by calling *next()* again, or it can be short-circuited by not calling *next()* anymore. In which case, that specific middlewares becomes the end of the pipeline. All dependencies in a middleware are resolved from the scope of the message that triggered the initial pipeline, and are released together with it.

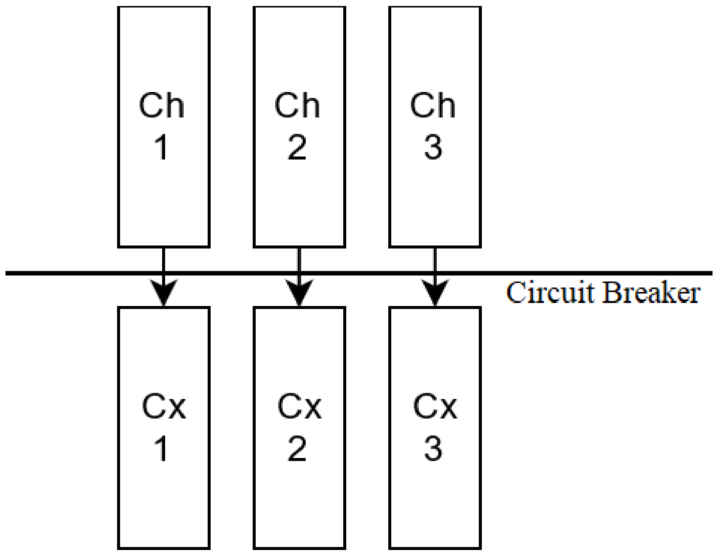
5.4 Circuit Breakers

Dependencies fail often however many times they fail with a reason. Request throttling or data throttling is enforced in many modern applications and often times they are used in SAAS services for subscription-based services where a different throttle would mean a different subscription tier. In the context of message-based communications, external dependencies can have a throttle in place or can simply reject requests because they have reached a limit. This would result in exceptions which mean immediate retries, and that does not help the system at all. Throttled failures always return some type of information for the consumer to know, when the action can be retried, typically under a header or a field *x-ms-retry-after*. For such situations, it is essential to be able to pause a certain service from processing for a specific amount of time.

NServiceBus employs a *private* circuit breaker which triggers on internal errors such as disconnections from the message broker and consumer reconnections. The circuit is not accessible from the consuming applications and as such the application must configure different retry techniques for situations where a dependency is unavailable for a specific amount of time.

In the case of SafeBunny the circuit breaker works a little different. It acts as a public tool which can sever a consumer connection from a channel or from the consumer perspective, stop consuming a certain type of message from any producers. On circuit breaker release, SafeBunny will attempt to reconnect the consumers to the channels and resume message processing.

The circuit breaker is built next to SafeBunny's internal topology management mechanisms and is part of it. When SafeBunny subscribes to the message broker it actually does so by opening the circuit breaker. Only consumer-channel connections are stopped in order to make the process as efficient as possible, none of the active topology is lost and no channel has to be re-opened. Re-activating consumers is a cheap process.



[Fig 5.4] Circuit Breaker point

The Circuit Breaker is exposed outside of SafeBunny's internals and can be triggered with a *Timespan*, (C# representation of a duration), after which it will automatically release.

Internally, the circuit breaker acts on internal connection errors and also in situations where the deduplication throughput of SafeBunny is hit [Ch 7.2], and the timeout is larger than expected.

5.5 Subscription pipeline

When a message is received from the broker, it goes through the processing pipeline, or the consumption pipeline. Without user additions the subscription pipeline consists of 3 components; execution component, delayed retry component and the poison message component. The execution and the delayed retry components execute in accordance to the initial subscription parameters. Each subscription to the message broker can be configured if it should retry either in memory or through the exchanged delay, how many times and with what delay. These components are part of the core processors and they cannot be changed or short circuited by the consumer.

Execution component

Since the pipeline is completely type agnostic, there is no generic information related to the type that is being processed besides the metadata present on the message context. The actual message generic is built and the handler is resolved inside the execution pipeline. Reflection [Ch 6.2.1] is used in order to remove any performance bottlenecks before the consumer handler is being executed. All of the user code execution is done in a *Polly* block, regardless of the amount of retries. A message configured to never retry will simply execute once the Polly block then leave. This means that the in-memory retry and any exception that could occur is handled inside the execution component.

Possible drawbacks come from this approach [Ch 6.2.3]. For larger delays between executions, a message that retries often could hold a processing spot on the instance for a period of time that is bigger than desired. Because all of the retries are done inside the pipeline, this means that the message does not go back to the broker nor does it get acknowledged. The alternative was to always send the message back to the broker and keep the throughput of messages high even in the case of exceptions. Sending a message back to the delayed exchange however disrupts the order of message which could cause further damage. With this reason in mind, delayed retries are the second method of resilience, and in-memory retries tank the performance hit first.

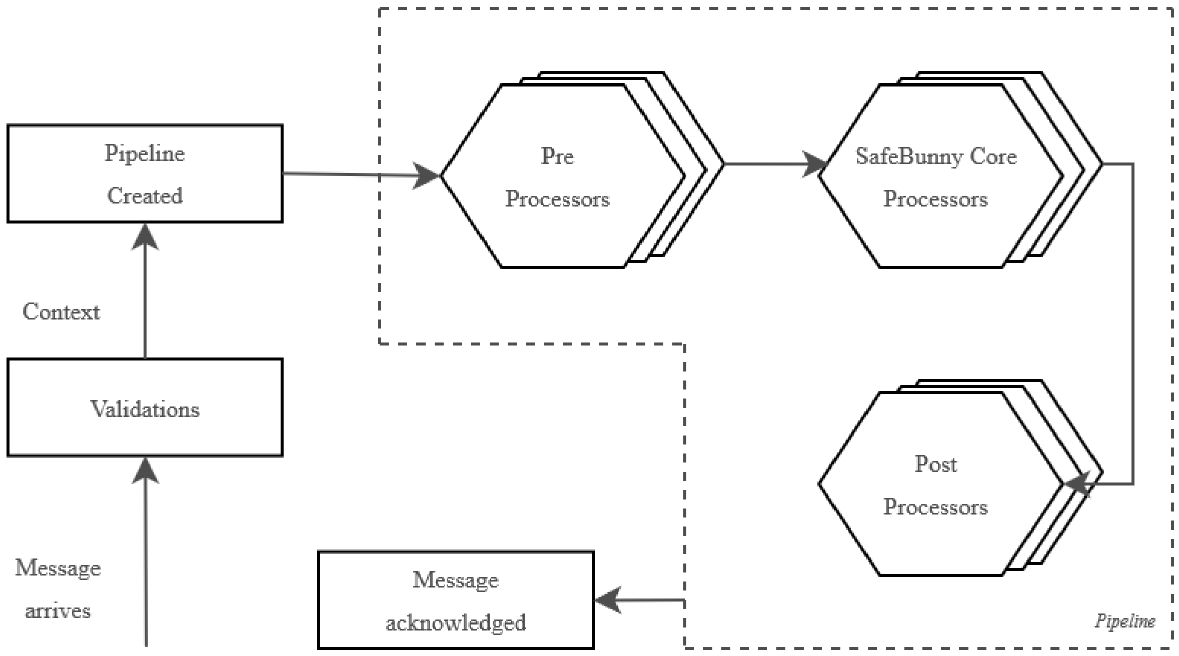
If a message fails the execution, including all the configured retries, the current context is marked as failed and it will be considered valid for a delayed try or a poison queue transition.

Delayed Retry Component

This component is responsible with sending a message that failed during in-memory processing to the delayed retry exchange. The component is skipped if the message is not marked as failed, there is no delayed retry configured for that specific subscription or all the retries have been exhausted. Each valid execution of the component sets the delay timer and increases the delayed retry count on the message properties. Deduplication properties are saved in the headers and they will be persisted between consumptions.

Poison Component

If a message is not processed during the in-memory component or the delayed retry component, it will end up on a poison queue. Afterwards, poisoned messages can simply be processed manually or left to be discarded by RabbitMQ after their expiration time passes.



[Fig. 5.5] Full subscription pipeline from the moment a message is received, until it is released

The main power of the pipeline approach is the extensibility the developer has. The pipeline can be short-circuited before, or after the core components of SafeBunny, middlewares can be added for logging, testing or subjugating the system to chaos theory.

Unless the pipeline results in an exception, the message is always acknowledged. This clears any potential problems related to memory leaks, ghost messages, or miscommunication with the message broker. Several exceptions in a row will send the message to the poison queue.

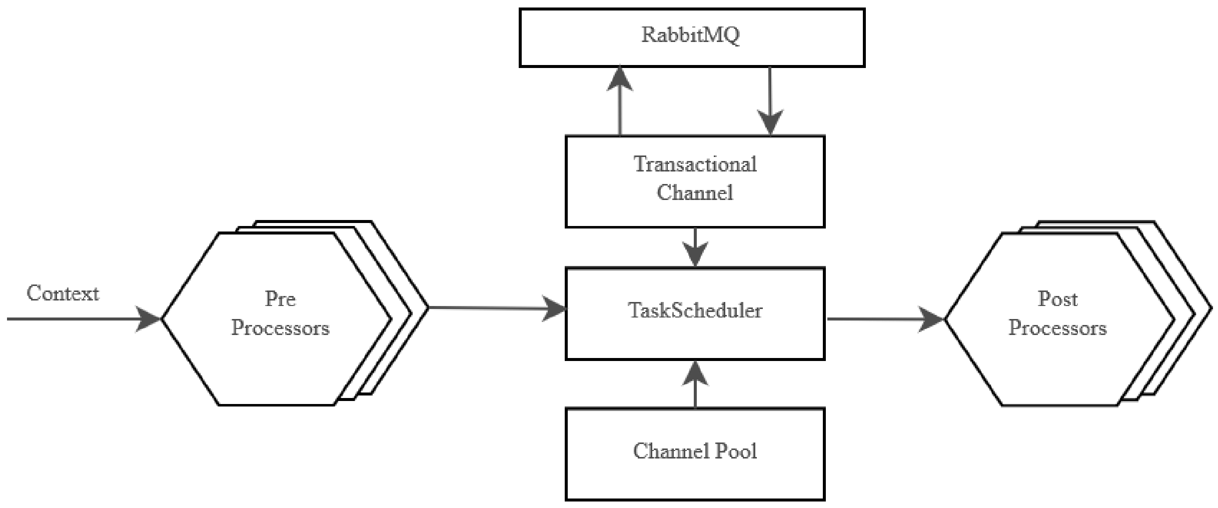
The validation step ensures that a channel is still open, the consumer is not hanging to a closed channel, the message type represents a valid message from the queue it was consumed from and not a stray message, the message body is not corrupted and all the required message properties exist.

5.6 Publication pipeline

Unlike the subscription pipeline, the publication pipeline has some key differences. Consuming messages from the message broker on multiple threads is perfectly safe with the RabbitMQ.NET library, however publishing from multiple threads on the same channel can be dangerous. These dangers and their fix are explained in [Ch 6.1].

Because publishing cannot simply be done concurrently in an efficient manner out of the box, the core processors in the case of publishing are a little different. There is only one single processor that triggers a publish through a custom task scheduler which handles the rest of the multi-threading issues. From there, a channel from a channel pool will publish the message and wait for the confirmation on another thread in the original task scheduler. If anything happens during the wait time an exception is thrown on the publish block, if not, the pipeline continues.

Continuation markers are also generated before publishing, based on the metadata present in the publishing context. The metadata itself is up to the user of the library to decide when a message represents the next component in a saga.



[Fig. 5.6] Full example of a publication pipeline, where the tasks scheduler takes a channel from a pool and uses it to publish the context to RabbitMQ.

By using transactional channels, possible mistakes such as publishing to a queue that does not exist or not publishing at all due to network interruptions, will result in exceptions which stop the handler at the publish step. Due to the nature of the pipeline, any pre processors will be still executed, and any distributed traces will take place. Because pre and post processors are basically hooks for pre and post-publish, the application can register various services such as application insights, custom behaviors, application wide validations or even stopping a publish on custom conditions.

5.7 Routing topology

There are several ways in which a topology can be configured on RabbitMQ. From exchanges and queues based on assembly to multiple consumers on the same queue handling different types of messages.

An exchange based on assembly would always route based on the namespace of a type, so this means that different services would have to share the types that they use to communicate. The only way to achieve this is either with NuGet packages or having a common project in the same solution. This design would drive the solution to get monolithical and any change in the type library would case an update to be necessary across the codebase, or messages wouldn’t be consumed anymore.

Another necessary feature is that multiple consumers can process the same type of message, a behavior encountered often in event-based communication. In the context of assembly-based exchanges, it would be possible however the exchange would be fanout. The case of queues receiving messages that cannot be processed appears. This excludes assembly-based topologies and fanout exchanges from the range of possible architectures.

Another way to configure the topology would be to have multiple types of consumers on the same queue and the dispatch to them fanout. Besides letting the user in control of the queue a message ends up on, this would increase the dangers of multi-threading conflicts as if the instances were to be scaled horizontally, all of them would process the same transaction. Removing the fanout aspect would solve this but then we are still left with multiple types of messages on the same queue. The problem that arises from this topology is that the non-deterministic factor of processing increases by a lot. There is no way to tell if a message type is hogging a queue and others are not being processed or less important types are being processed before higher-priority ones. A fix would be to give each message type a priority, and then RabbitMQ would automatically order the types of messages on the queue. This translates into an extra burden on the message broker which is undesirable, and also the high number of “patches” can tell us that this topology is not the best that can be done either.

SafeBunny chooses a different way to configure the broker topology and completely abstracts it from the user. He does not know the name of the queues, exchanges, or routing keys. He does not get to choose to what exchange a message is published and what routing key it uses.

Connections, channels, and consumers are abstracted as well, and they are not part of any public interface. The best approach for SafeBunny is for the library to be in full control of routing and topology, enabling the user to focus the entire codebase on business logic.

The only visible part of the topology that a user sees, is during application initialization, as the user must choose the *node* that he wants to consume a type of message from. A node represents the string identifier of a processing node.

```
subscriptions
    .FromNode("billing")
    .Consume<InvoiceCreatedEvent>()
```

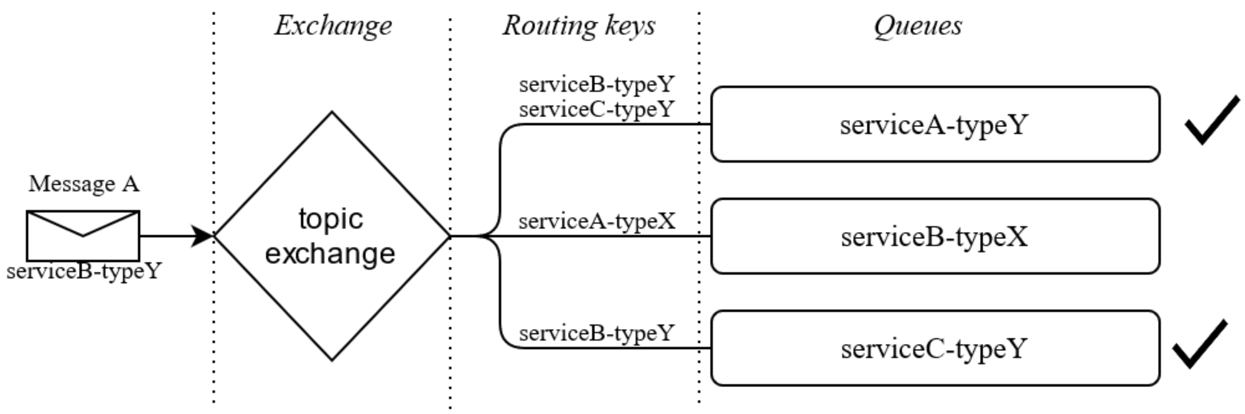
[Fig. 5.7] In this code snippet, the current service wants to consume ***InvoiceCreatedEvent*** *from the* ***billing*** *service.*

SafeBunny uses the name of the node in order to build the necessary links in the topology and the routing itself. Using *Infrastructure-as-Code*, any part of the topology is always rebuilt, unless it already exists, when any instance starts. SafeBunny declares a delayed exchange, the topic exchange that is used for routing, all the queues that are to be consumed by the current instance and then assigns all the routing-keys to the queues. Any circuit breaker stops do not re-trigger this operation, it only happens on application startup.

The routing itself consists of the name of the node which publishes/subscribes and the name of the *type* that is being consumed or published.

Each Queue that is created bears the format *{consuming-node}-{type}*. SafeBunny automatically binds to this queue the service where this message type comes from, *{publishing-node}-{type}*, and the queue is bound to the default SafeBunny exchange. When a node publishes a message, the message bears the routing key built from the originating node and the type of the message *{publishing-node}-{type}*. This means that a message type can be simultaneously routed to multiple consumers regardless of how many instances of that instance are up, and the message consumption will be separate from other services consuming the same message type. This gives both performance benefits and separates the concerns perfectly. It also enables a more fluent event driven architecture if those are the wishes of the application.

With this approach, the routing effort is placed on the message broker, however in a cluster of RabbitMQ nodes, the effort is negligible, the routing complexity scales with the number of queues and the number of queues will scale proportionally direct with the number of services and the numbers of message types consumed.



[Fig. 5.8] Routing result in a topology with 3 services and 4 subscriptions. Service A and C are both subscribed to Service B’s message type Y and so they both get the message.

6. Performance considerations

Because SafeBunny is a multi-threaded, message-based communication library, pretty much the entire library represents a hot path for any use-case. Most of the library code is executed before the user code is hit, and then the library code is hit again when the user part ends in order to finish the consumption process. Multi-threading is important, since a process wants to consume as many resources as possible in order to consume as many messages as possible, however it is desirable for those resources to be consumed by the least amount of library code as possible. This indirectly results in higher message throughput, but higher throughput means more dangers that something can go wrong between threads that are simultaneously publishing or consuming messages. All of SafeBunny is generic, the code itself does not know explicitly what type of message it is consuming, or what are the responsible message handlers, or how many modules there are in a pipeline. This means that a lot of reflection happens throughout the library. The last extremely important aspect is the serialization and deserialization of messages. The process of serializing and deserializing happen in 100% of the messages that are consumed through SafeBunny.

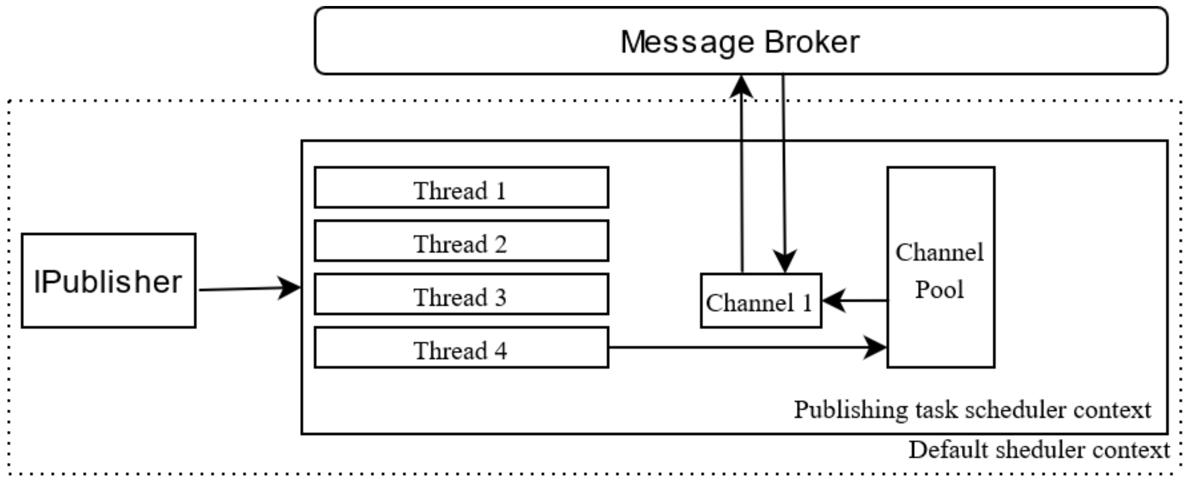
6.1 Thread safety dangers

The biggest problem between threads happens when the process hosting SafeBunny wants to publish and consume at the same time. All the communication between the process and the message broker happens on *Connections*. In order for the TCP packets to not interfere with each other, the RabbitMQ teams strongly recommends that Connections should be used one way. For this reason, SafeBunny opens two connections, one for publishing and one for consuming.

The second issues arises when a channel, or multiple channels from the same connections attempt to do the same process twice. Subscribing is generally safe in multi-threaded contexts [17], and SafeBunny offers a configurable prefetch size, allowing the application to consume a number of messages at the concurrently. Publishing, on the other hand, executed from the same channel on different threads simultaneously is not safe in .NET. It can result in packet errors and corrupted messages arriving at the message broker.

Multi-threaded publishing is a must that has to be fixed, as it is essential for the throughput that also publish at the end of their lifecycle. In practice, several applications either lock a channel across all threads, until it has finished publishing, or they consume just one message at a time, which ensures that only one message is being published at a time. Other solutions include spawning a channel every time a message has to be published or even connections. Channel churning, or spawning a large number of connections puts an unneeded load on the message broker and reduces the overall throughput at the infrastructure level. For this reason, SafeBunny uses channel pools and long-lived connections in all use cases.

In order to tackle to issue of multi-threaded publishing, SafeBunny employs a custom Task Scheduler and a publishing channels pool, that are different from the consumption ones. Each channel in the pool is responsible of publishing and confirming the publish of a single message, at which point the channel's job is considered done and it is returned to the pool. From the pool, a channel is picked at random when one is needed, using C#'s *BufferBlock* [18]. If there are not enough channels, the application can either configure a larger number of channels or wait for one to be returned. The task scheduler enters the flow when the pipeline invokes a publish. Publishing is done by sending an anonymous function, or a *Func<>* in the case of C#, to the Task Scheduler. Using the available thread pool, the task scheduler will invoke the publish on a separate thread and marshal back to the original context when it has finished. By limiting concurrency number inside the task scheduler, we can control the maximum concurrency level of outgoing messages.



[Fig. 6.1] Example of the custom task scheduler with one thread in use that is publishing to the message broker.

The *BufferBlock* structure is part of the new .NET Core 5.0 SDK, present in the *DataFlow* namespace. It offers very lightweight and low control of data inside the buffer. It works like a Conccurrent Queue, which is used by NServiceBus, however it has built in support for asynchronously waiting when the buffer is empty. SafeBunny makes use of *Interlocked* [25] in order to track down what happens inside the asynchronous buffer. It keeps count of the available channels and the total number in channels. If there is room available, it will create more channels in the pool and if not, wait with the built-in system for a channel to be returned by the task scheduler. The maximum concurrency can be configured with the *MaximumPublishConcurrency* field.

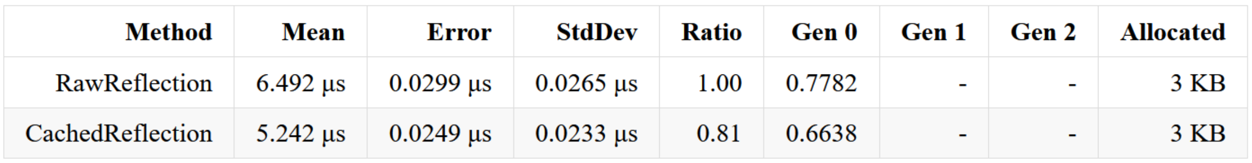
6.2 Hot paths

From the moment a message is received and user-code is hit, there are several places where optimizations can be brought in order to reduce the processor time that is spent in SafeBunny code. Because SafeBunny can handle multiple types of messages in the same process, the first decision that has to be made is the selection of the *Consumer*. Because the consumer is responsible with acknowledging or rejecting messages, it is the first validation in the pipeline. SafeBunny stores consumers in dictionaries, which in C# have *O(1)* access times. A message is immediately discarded if the consumer is offline or invalid, cases that can happen given the nondeterministic nature of the infrastructure.

Next comes the deserialization of the message body itself, which depending on the message size and complexity can be the most time-consuming factor, pipeline building, pipeline execution and the decision between acknowledging and rejecting. Everything after deserialization heavily relies on Reflection, which is a C# property that can resolve and execute types at runtime, based on metadata, keeping them abstract at compile time.

6.2.1 Reflection

Reflection is used in order to resolve the message context in order to create the pipeline, pipeline components, message handlers and various functions in SafeBunny. In order to offer a comfortable interface, all public contracts of SafeBunny are generic. This means that SafeBunny has to work extra in order to bring the type anonymity to a generic layer. SafeBunny does so by saving metadata from the moment a subscription is declared (Fig 5.7). This simple message type and node information is expanded into metadata regarding the consumption handlers implemented in the user code, which are automatically scanned, message context types, method and constructor information. All this caching allows SafeBunny to perform faster and execute complex tasks such as resolving types and invoking them without traversing the reflection tree at every message event.



Method	Mean	Error	StdDev	Ratio	Gen 0	Gen 1	Gen 2	Allocated
RawReflection	6.492 μs	0.0299 μs	0.0265 μs	1.00	0.7782	-	-	3 KB
CachedReflection	5.242 μs	0.0249 μs	0.0233 μs	0.81	0.6638	-	-	3 KB

[Fig 6.2] Cached reflection enables a 19% performance improvement.

Once all of the necessary reflection metadata is cached inside SafeBunny, the only complicated tasks left are transforming types from the strict *Type* to a generic <*T*> in order to present them in public interfaces.

6.2.2 Fast Serialization

Serialization and deserialization play an important role in the pipeline since it can go as far as break messages. Because it is used in consuming messages and publishing, it has to be fast but also reliable. Large messages can take precious time to serialize and deserialize and also can occupy more memory than necessary. For this reason, besides offering a default serializer, SafeBunny exposes a public interface with which developers can implement their own serializers that are fit for their application.

There are many options for serializers and each perform extremely well in very specific contexts. In the case of message-based system, binary, json and proto serializers are all good choices depending on the messages that are being consumed and sent.

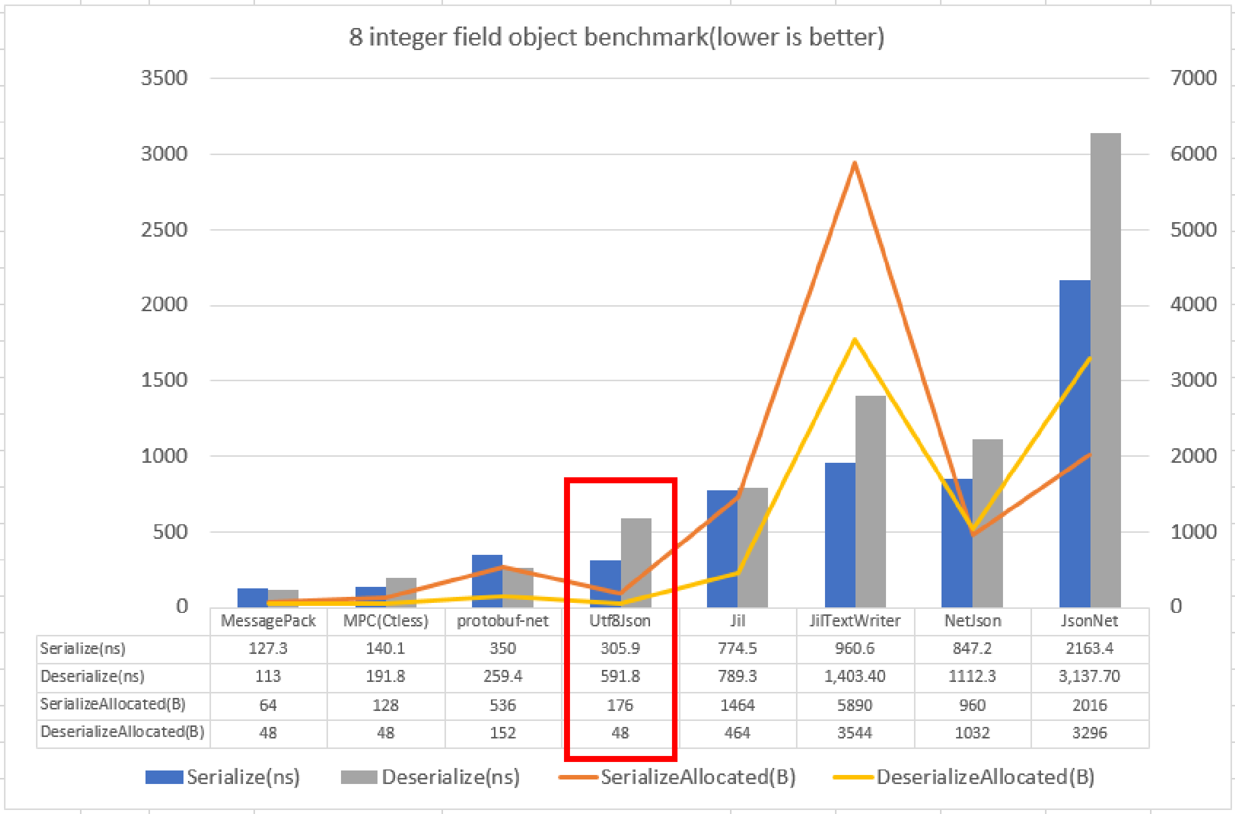
SafeBunny's goal is to reduce the necessary knowledge that a developer must have in order to use a message broker and also solve all the concurrency and possible dangers that come with a message-based infrastructure. The default serializer was chosen with this in mind.

One of the fastest serializers and a global standard is the proto-buffer [19], developed by google. In the context of C#, proto-buffers replace the keys of objects with short, unique integers and then compress the data into binary streams. This makes proto-buffers the most efficient for transferring data over the network. In the context of SafeBunny, this comes with a major drawback. Proto-buffers pollute domain models with forced attributes in order to achieve the proto buffer format. With SafeBunny we want to avoid as much as possible code pollution and keep any models clean and representative of the domain logic that they are solving. Another possible issue is that inheritance and generic messages become complicated to express in proto attributes.

Inside .net, there is a Binary serializer [20]. It uses several caching and reflection mechanisms to achieve high speed binary streams with performance greater than proto-buffers but at higher memory footprints. Being native .net, it comes with the advantage of no third-party libraries along the performance however the binary serializer has another major disadvantage; and models that are meant to be deserialized must have matching assemblies at the moment of their serialization. What this means is that the models must be shared otherwise the deserialization will throw an exception. The only way to achieve this is by either sharing the models through NuGet packages or being in the same solution. Both ways break the convention of microservices since they become dependent and not individually deployable after changes to core domain logic.

The following two options are NewtonSoftJSON [21] and .NET JSON Serializer. Both represent the same thing as the .NET JSON Serializer is Microsoft's attempt with .NET 5 to merge the NewtonSoftJSON style if serialization inside the core .NET 5 codebase. NewtonSoftJSON is used widely in many production instances and accepted as the standard JSON serializer in .NET for any distributed use. JSON is easy to serialize and deserialize without having references of said models and also allows partial deserialization, which can be of great advantage in the context of message-based systems, when a consumer only needs a part of the message.

The last and the choice of SafeBunny is Utf8Json [22]. It implements utf8 based encoding and it offers a very large array of possible customizations which can help for very niche use cases that applications may have in their domain models. Besides the greater flexibility compared to NewtonSoftJSON, Utf8Json changes the way in which serialization is handled by using a very low approach with automata and IL code.



	MessagePack	MPC(Ctless)	protobuf-net	Utf8Json	Jil	JilTextWriter	NetJson	JsonNet
Serialize(ns)	127.3	140.1	350	305.9	774.5	960.6	847.2	2163.4
Deserialize(ns)	113	191.8	259.4	591.8	789.3	1,403.40	1112.3	3,137.70
SerializeAllocated(B)	64	128	536	176	1464	5890	960	2016
DeserializeAllocated(B)	48	48	152	48	464	3544	1032	3296

[Fig. 6.3] (github user neuecc - June 9, 2018 on Utf8Json [23])

It can be observed that Utf8Json brings the best performance compared to other possible serializers in the context of SafeBunny

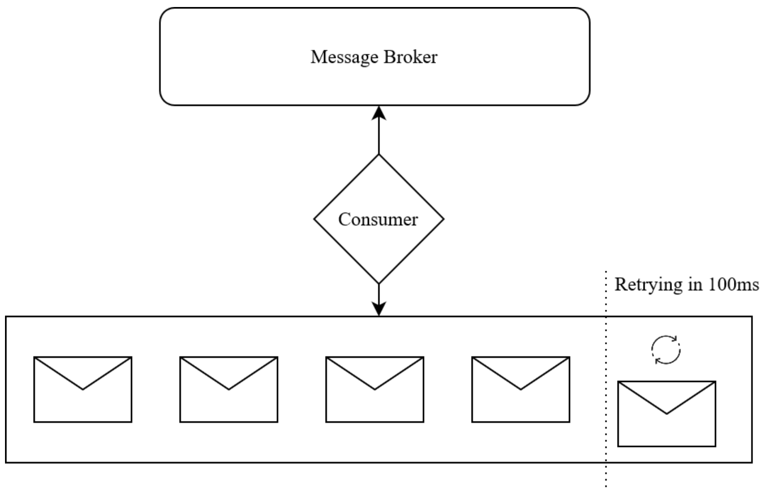
Developers can implement the *ISafeBunnySerializer* interface and use their own implementation of any serializer. The contract offers <*T*> for serialization, the entire object including the type and a *(byte[], Type)* tuple for deserialization, the byte array along the expected output type. A stream was not offered because RabbitMQ.NET already transforms any data streams into byte arrays at the end of its pipeline.

By using a JSON format, it makes it easy to plug-in any other microservices that are not necessarily using SafeBunny or even written in C#, but want to consume messages off the message-broker that are published by a SafeBunny powered service.

6.2.3 Retries and RabbitMQ prefetch exhaustion

In the model of using retries, two issues appear. First, in memory retries are concurrent with other messages, meaning that while a transaction is retrying, another subsequent retry will execute in the same time, and lead to even more retries before the first one succeeds. Second, because delayed retries are sent to the bus and back after a duration of time, it means that the same behavior for in memory retries applies, a chain of transaction will be sent to the broker simply as a consequence of the first one failing. This, is not necessarily a bad thing since transactions are eventually going to converge and not affect the application. The delicate part is handing in memory retries correctly.

As explained in [Ch 4.3] each consumer has a set prefetch count, a maximum number of messages that can be processed concurrently. This means that if a message is hanging or is forcefully hanged in memory because of retries, it results in one prefetch slot being frozen. Having a frozen prefetch slot drastically reduces performance, assuming that a service has a prefetch of 10, a frozen slot results in a throughput reduction of 10%.



[Fig 6.4] With a prefetch of 5, one message freezes a prefetch slot for 100ms, resulting in a 25% throughput reduction for 100ms.

An alternative to in memory retries is sending the message back to the broker. The retry time is instantaneous and there is no theoretical throughput reduction. In practice, the same message is still being processed instead of a new message, and there is an extra load on the message broker to maintain message ordering on message rejection. The downside to returning messages to the broker is the loss of timed retries. It is impossible to control the time for when the message itself will be retried, which can be essential when there is a small race condition or a database could be down for any milliseconds. The retry time becomes the latency of publishing the message, arrival to the broker and sending it back.

Another way in which messages can be retried is using any job schedulers, for example *HangFire* or *Quartz.NET*. The message is saved in a persistent store, the prefetch slot is released and then the message is fired back whenever a free slot opens and the retry time expires. This however cannot be done in transactional channels since in order for a prefetch slot to be open, a previous message must either be acknowledged or rejected. Disabling the transactional properties of RabbitMQ can result in messages being lost seemingly at random over the network, since there will be no more confirmation between the broker and the consumer regarding message arrivals, acknowledgments or rejects. Such a system can be beneficial where message loss is acceptable, however SafeBunny targets systems where transactions and message durability over the network take priority.

SafeBunny handles in memory retries using *Polly*. Each retry can be configured with a *retry time* and a *retry count.* Retries are executed in memory using *Polly's scheduler*. Having been proven in production instances, SafeBunny relies on Polly for efficiently coordinating tasks and threads from prefetch slots that are being frozen and awaiting them with minimal performance loss.

Depending on the application in question and its dependencies, retry timers can be optimized in such a way that the frozen slot issue is minimized. If a dependency failure occurs, any subsequent messages will fail as well and the throughput will hold still regardless. By freezing the slot until the expected dependency recovers, no resource is lost and no message is processed in vain.

Retries can be disabled all together for types where such a mechanism is not needed, only delayed retries can be used or none at all. By combining the retries with the circuit breaker the application can act resilient in front of any type of exception, and gracefully retry messages away from the calling service.

```
subs.FromNode("billing")
         .Consume<InvoiceCreatedEvent>(
             10,
             RetryStrategy.From(2, TimeSpan.FromMilliseconds(100)),
             RetryStrategy.None);
```

[Fig 6.5] The subscription is configured with a prefetch size of 10, two in memory retries with a delay of 100 milliseconds between them, and no delayed retries enabled.

Because all in memory retries execute immediately in a *Polly* block in case of an exception, it is desirable for scoped or transient instances to be idempotent towards multiple calls in the same message event. The pipeline does not reinitialize components between retries so the implementing application must ensure that there is no danger in executing the same handler multiple times for the same message entity. Reinitializing the pipeline would mean more wasted memory than and processor time than necessary.

7. Deduplication

Once all the practical issues about a message broker are taken care of, reliably publishing, consuming, retrying exceptions, being robust against unavailable dependencies and organizing the code in such a way that it is easy to maintain and deliver business value, the last and most important part becomes the integrity of the data that is being transmitted over the infrastructure.

Having corrupt, invalid, or untrue data being received in a service once the application relies on it to conduct costly customer actions, can affect the application negatively even if it happens in under *1%* of cases. Such issues are often latent and only appear once the application gains traction, the traffic is higher and certain limits are hit. In general systems, such an issue is for example a domain validation that does not have strict database checking. An email is checked for uniqueness at the application level and then inserted freely in the database. Once traffic kicks in, it is very possible for two requests to have the same email but the application will validate them both.

These issues have a different form in distributed systems and they show up in duplicated messages, transactions that do not execute however they are believed to be executed, dependencies failing to publish events that other services depend on and executing transactions multiple times when they should not.

In order to solve these issues, SafeBunny proposes the combination of two methods. First, in the case of duplicated messages to not be processed simultaneously or again at a later time, on the same processing instance or in a distributed manner, SafeBunny centralizes the correlations of a logical processing service and offers a way in which services can execute transactions through this mechanism. In other words, each transaction gains an identifier which is unique immediately in the scope of all the current instances of said service. If it were to execute again, or at a later time, SafeBunny will cancel the transaction before it is executed.

From the first mechanism we can observe the main flaw for which it will fail. Every time a message is published, along the correlation id it will gain a unique message id. It is impossible to ensure that a transaction did not happen before if every single message gains a unique id. Saving the correlation id instead is also an issue if a service can process, and definitely will process multiple types of events that are created from the same initial action. In order to solve this, SafeBunny created deterministic message id's, which ensure that if a message is published from a specific context, it will always carry on the same id, no matter how many times the action is repeated.

By combining the two methods together, we can ensure that once an action begins, every unique transaction and message published is tracked inside the logical process that is handling it. Executing a transaction and publishing a message will always produce the same results. With this, any service that consumes a message can verify if it was not already consumed, effectively reducing any duplication errors to 0.

7.1 Deterministic Message Ids

Generating deterministic message ids in a reliable way can be quite difficult, SafeBunny could save timestamps and generate ids with those as seeds, however there is an extra requirement for these id’s to be always valid. A saga is constructed of several components and each components wants to effectively publish at the end of it, the continuation of that component. SafeBunny creates *markers*.

A marker will reflect the number of that event in that saga, and increment it for the next. Based on this marker and the correlation id of the action, a message id is generated for the next message in line. Even if an event is consumed by multiple services, theoretically they will both have the same continuation marker however they represent different things and end up in different logical nodes, continuing a different branch of the saga, on the same correlation.

The next hard cap that a continuation marker could have is the size of it, it is not expected that a saga could be built of more components than the max value of an integer, however using numbers as unique ids for a message is generally not a good idea. For this reason, the message is constructed from the continuation translated in a 25-numerotation base, and then applied over the correlation id, which spans 32 characters. Each id in SafeBunny is a *Guid* generated by .NET. This ensures that the id still looks like an id however it is always unique in the scope of the logical processing node and completely deterministic based on the correlation id.

Correlation:	*db19dadf-39d7-4529-836d-cb89e97bfb48*, Marker: 27
Resulted Mask:	*00000000-0000-0000-0000-0000000000ab*
Resulted Message Id:	*db19dadf-39d7-4529-836d-cb89e97bfbab*

[Fig 7.1] The generation of a message id

In Figure 7.1 we can observe the generation of a message id. The initial action generates a correlation id, and each saga component will start increasing the continuation marker. The continuation marker is transformed in a 25-base representation of the decimal version of it and then applied over the correlation Id.

In this way, there is the certainty that message ids will never be repeated in the context of an action, across an entire saga and they will always be deterministic.

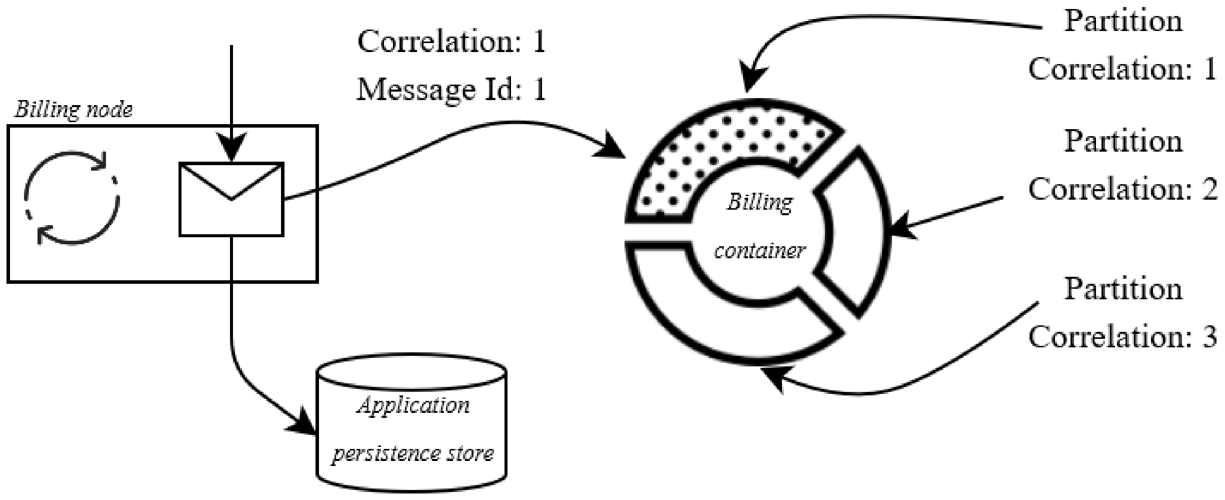
7.2 CosmosDB as the transaction store

With message ids safely deterministic at publishing across the system, the next part is finding a suitable persistence store to centralize transactions across a single logical processing node. The drawback of typical stores is that in an acid database, an entire column is frozen during a transaction, which is not efficient if that column contains more information that needed to execute the transaction. Another requirement is that the existence of a transaction must be validated on both the software layer and the infrastructure layer, in order to avoid the tightest race conditions and also validated in an efficient manner. Collections for correlations can be created in NoSQL but that will very rapidly exhaust the resources of such a store.

With CosmosDB we can abuse the partitioning aspect of a container, in order to only execute transactions and validate unique keys against the correlation which we are interested in. This reduces transaction times and the scope of validations down to the subset of saga components inside a single logical node.

A partition key represents the correlation id of an action and the key becomes the message id. Each logical node becomes a container with a dedicated throughput storing all of the transactions executed across all instances.

Validating a transaction comes down to reading the messages in a partition, and checking if it already exists on the software layer, and then trying to create the unique key on the database level. Because the transaction scope is tiny and the reads are less important that the writes, the consistency levels of the CosmosDB instance do not matter much, and can be set on whatever level brings the most performance in the context of a specific application using SafeBunny.



[Fig 7.2] Example of a message from correlation id: 1 with message id: 1 being validated against all the messages previously processed with correlation id: 1, completely separate from all the other partitions

8. Performance

In order to gauge performance in a close-to-real situation, we will use *Integration Benchmarks*. For these runs, the components are not tested in isolation anymore and the entire library directly connects to a RabbitMQ cluster.

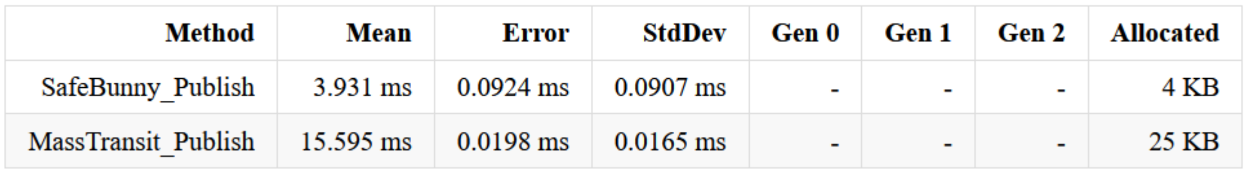
For integration benchmarks, the system is the same as described in [Ch 3.4] with a new addition. The RabbitMQ cluster used in the benchmark is formed of 3 nodes, in total 16GB of RAM and 4 Ryzen 2700x CPU cores.

For all benchmarks, a run is built of three stages, warmup – 8 iterations, actual – 16 iterations, result – 16 iterations. Each iteration will execute the benchmark 128 times, and clean the memory between stages. This means that it simulates an action concurrently 128 times.

8.1 Publishing

Important notes in the case of publishing. A publish is considered reliable when the message broker confirms that a message has been accepted and routed correctly. Applications have to be designed in order to wait and throw an exception if a message in not routable.

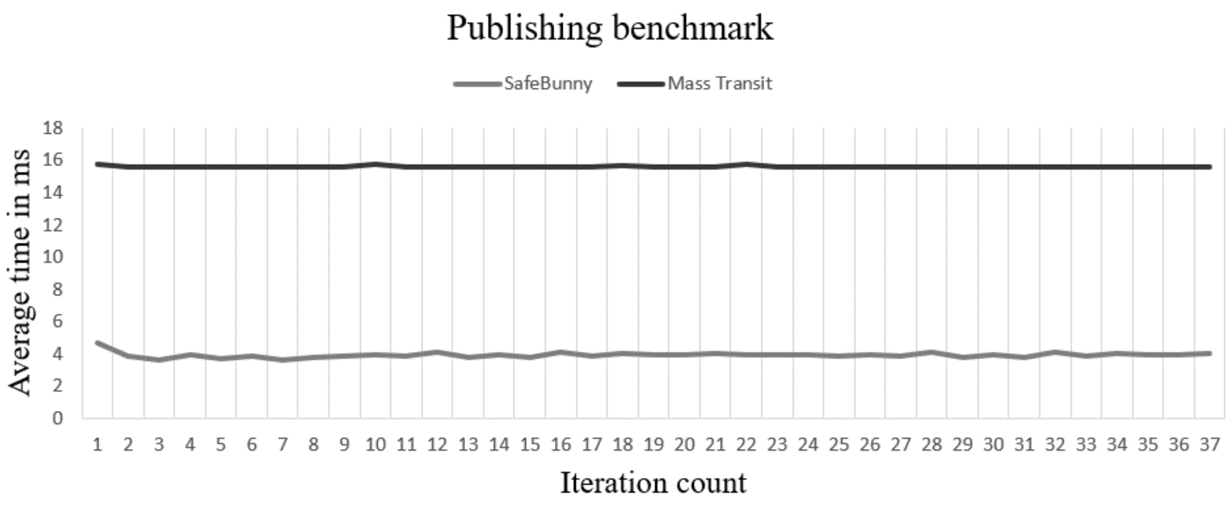
For publishing, we are comparing SafeBunny with Mass Transit, executing the entire library code, simulating a live application. Both are configured to publish durable messages to durable queues, with one difference. Mass Transit confirms that a message has arrived once an exchange routes it, and SafeBunny confirms when a message reaches a queue.



Method	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
SafeBunny_Publish	3.931 ms	0.0924 ms	0.0907 ms	-	-	-	4 KB
MassTransit_Publish	15.595 ms	0.0198 ms	0.0165 ms	-	-	-	25 KB

[Fig 8.1] Mass Transit compared to SafeBunny in simple publishing, with no state machines.

In Fig 8.1, SafeBunny is limited with a total of 8 publishing threads, while Mass Transit manages the threads using a different internal mechanism. A benchmark measures the total time of publishing 1 message to the message broker, publishing thousands of messages in the process. Both libraries reach equilibrium as the publishing channels are exhausted and they have to start waiting for available resources. At the end of the benchmark, both libraries publish 3216 messages to the broker.

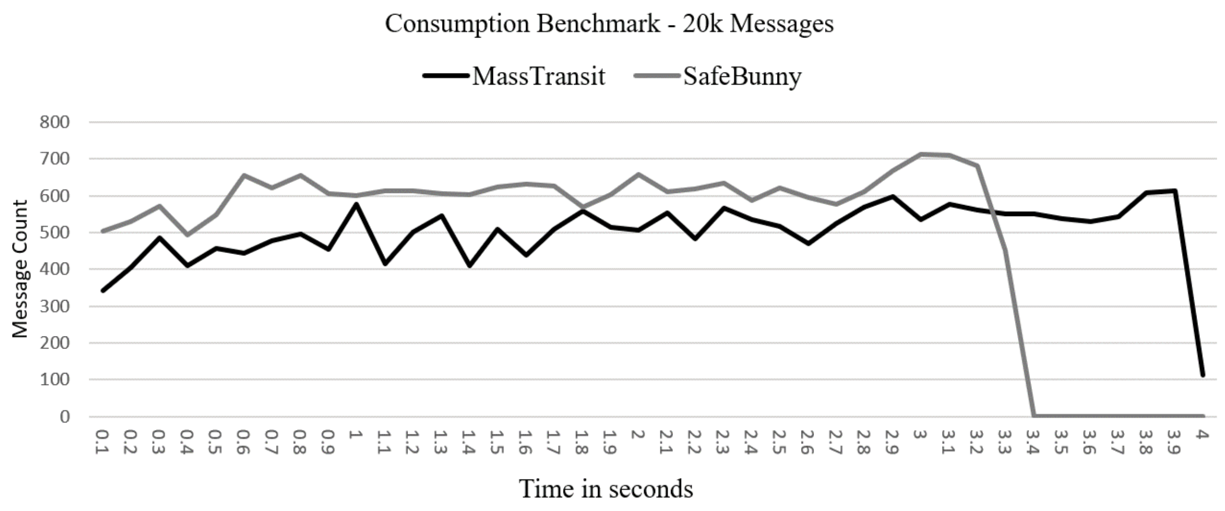


[Fig 8.2] Time spent by both libraries over all the benchmark iterations

The main difference in publishing between the libraries is the way they handle the threads behind the scenes. SafeBunny manages a channel pool asynchronously which means that in the case of starvation, the timings will fluctuate, while Mass Transit is a lot more robust and does not fluctuate.

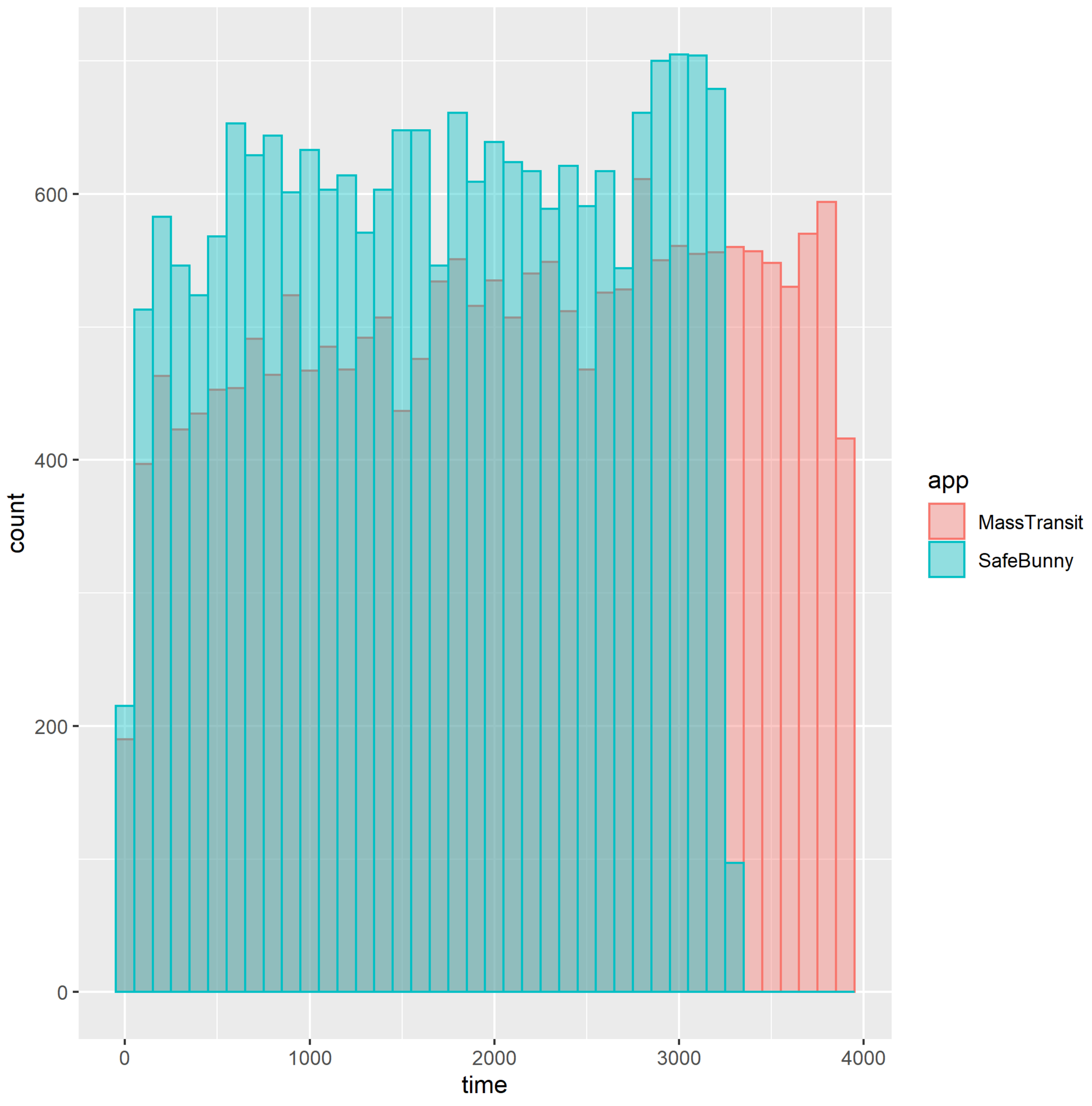
8.2 Consuming

In order to benchmark the throughput of the two libraries, the approach needs to be a little different. Measuring the entire consumption pipeline is difficult in the case of Mass Transit because those are all internal classes that cannot be instantiated easily. To analyze the throughput, each library will be used inside a live application that will consume 20k messages off the message broker.



[Fig 8.3] SafeBunny and Mass Transit consuming 20k messages from the message broker

The benchmark was conducted with a strict prefetch count of 10. Both libraries could only process 10 messages at a time, so the bottleneck was strictly processor time spent on library code. All the consumed messages were published ahead of benchmark on the message broker in order to rule out outside interference and had identical values. All the queues were durable and consumer acknowledgements were turned on. In the results of Fig 8.3 we can observe that the overall throughput of SafeBunny (~*600 messages/0.1ms*) is a little higher than that of Mass Transits (~*500 messages/0.1ms*). A better difference can be observed in Fig 8.4.



[Fig 8.4] Histogram showing the number of messages processed by each library in bins of 100ms. SafeBunny averages 600 msgs/0.1ms and Mass Transit 500 msgs/0.1ms

9. Conclusions

With today’s standard of distributed systems, a library that wants to fulfill every single aspect of a message-based communication library must check many items.

Starting when the application is born, time-to-market becomes the most important factor in choosing a library. Being able to push features as fast as possible while maintaining a decent code base. SafeBunny offers this using easily testable, completely decoupled middlewares and handlers, that can integrate in pretty much any architecture of services.

Once the application gains traction, performance starts becoming a matter of interest, it is faster and cheaper to optimize the code instead of hosting a second instance of a service. For this reason, the library must be as lightweight as it gets and optimized across all the lines of code. All possible optimization spots have been benchmarked and optimized in order to fulfill this goal.

Distributed tracing is important in order to track down distributed transactions. A transaction may seem fine and then fail two or three services later. By linking up middlewares at the beginning and end of SafeBunny’s pipeline, complete tracing is obtained at every single consumption and publication of a message.

After the growing phase, reliability and customer trust in the actions performed against a service become the deciding factor of an applications growing rate. To increase the reliability and trust factors of a service as much as possible, SafeBunny validates every single transaction with an external data source and deduplicates any messages it detects.

Message Brokers are a difficult thing and SafeBunny successfully abstracts RabbitMQ away from the developer, taking care of all the tedious tasks of durability, retrying, confirming messages arrive, surviving application restarts or unexpected shutdowns, message routing and overall topology of a distributed system.

A message-based library must pay attention to all the key points in developing an application itself, with no applications, the library holds no value.

10. Future work

In the case of distributed message systems, a piece of software like SafeBunny is never done. Distributed message frameworks have to continue evolving along with the applications, stay in touch with the latest and most powerful tools in the industry and keep providing the best service.

For future work, there are several aspects which can be improved:

- Transaction store options:

CosmosDB can be expensive, and not the easiest to get into if the application is small, more stores could be supported even if the drawbacks are there, for example a SQL store.

- In memory transport:

For testing purposes, it’s not always reliable to connect to the message broker. Applications desire to fully test their functionality without publishing messages to a public infrastructure inside the distributed scope. With this, a in memory message broker could effectively solve all these needs.

- A third way of retries:

Enable to application to return messages to the broker as a form of retry. This would not be much of a feature however it would be flexibility in the case of applications where throughput is more desired than durability

- Live watchdog application

A simple portal that connects asynchronously to all processing nodes that are up, can give a very nice over-all view of a distributed system. Health checks, message rates, failed transactions, error rates are all important aspects of a system which the default RabbitMQ portal does not provide.

- Real-Time encryption

Along the portal, SafeBunny can host an additional server which would hold key rings. Each logical processing node can receive unique their own key ring and the public keys of others. The keyring server would be responsible with automatically refreshing these keys at a specific time period and managing their lifetime throughout the lifetime of the application. With full cloud support, Azure Key Vaults [24] could be used for this.

- An integration with JavaScript

It is possible to communicate to the message broker from a separate language all together, however it would lack all of the SafeBunny features. It would be nice to integrate SafeBunny withing a JavaScript library, so clients could start sending messages directly from their client web apps.

Bibliography

1. [ABL] Challenges with distributed systems, [*https://aws.amazon.com/builders-library/challenges-with-distributed-systems*] [23.06.2021]
2. Microsoft Orleans at-most-once delivery [*https://dotnet.github.io/orleans/docs/implementation/messaging_delivery_guarantees.html*] [23.06.2021]
3. Exactly-once deduplication [*https://exactly-once.github.io/posts/token-based-deduplication/]*[23.06.2021]
4. RabbitMQ.NET [*https://github.com/rabbitmq/rabbitmq-dotnet-client*] [23.06.2021]
5. Polly [*https://github.com/App-vNext/Polly*] [23.06.2021]
6. Azure CosmosDB [*https://docs.microsoft.com/en-us/azure/cosmos-db/introduction*] [23.06.2021]
7. Partitioning in CosmosDB [*https://docs.microsoft.com/en-us/azure/cosmos-db/partitioning-overview*] [23.06.2021]
8. Microsoft's request pipeline [*https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-5.0*] [23.06.2021]
9. RabbitMQ Connections [*https://www.rabbitmq.com/connections.html*] [23.06.2021]
10. RabbitMQ Channels [*https://www.rabbitmq.com/channels.html*] [23.06.2021]
11. RabbitMQ Consumers [*https://www.rabbitmq.com/consumers.html*] [23.06.2021]
12. RabbitMQ Delayed Exchange [*https://github.com/rabbitmq/rabbitmq-delayed-message-exchange/*][23.06.2021]
13. NServiceBus RabbitMQ Plugin [*https://github.com/Particular/NServiceBus.RabbitMQ*] [23.06.2021]
14. Convey RabbitMQ Plugin [*https://github.com/snatch-dev/Convey/tree/master/src/Convey.MessageBrokers.RabbitMQ*] [23.06.2021]
15. Mass Transit [*https://github.com/MassTransit/MassTransit*] [23.06.2021]
16. Polly's Circuit Breaker [*https://github.com/App-vNext/Polly#circuit-breaker*] [23.06.2021]
17. RabbitMQ Concurrency considerations [*https://www.rabbitmq.com/consumers.html#concurrency*] [23.06.2021]
18. BufferBlock [*https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.dataflow.bufferblock-1?view=net-5.0*] [23.06.2021]
19. Proto-buffers [*https://developers.google.com/protocol-buffers/docs/overview*] [23.06.2021]
20. Binary Serialization [*https://docs.microsoft.com/en-us/dotnet/standard/serialization/binary-serialization*] [23.06.2021]
21. NewtonSoftJSON serializer [*https://www.newtonsoft.com/json*] [23.06.2021]
22. Utf8Json [*https://github.com/neuecc/Utf8Json*] [23.06.2021]
23. Image posted by user neuecc on Utf8Json [*https://user-images.githubusercontent.com/46207/30883721-11e0526e-a348-11e7-86f8-efff85a9afe0.png*] [23.06.2021]
24. Azure KeyVaults [*https://azure.microsoft.com/en-us/services/key-vault/]*[23.06.2021]
25. Interlocked [*https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked?view=net-5.0*][23.06.2021]