City University London
MSc Software Engineering (part-time)
Project Report
2015

Implementing operator state checkpoint based fault-tolerance in Apache Storm

Toby Stableford
Supervised by: Dr Evangelia Kalyvianaki
8th January 2015

1. Declaration

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed:

2. Abstract

Distributed stream processing systems such as ***Apache Storm*** enable near real-time analysis of the continuous data streams generated by today's modern enterprises. Cloud computing environments are well suited to needs of distributed stream processing, however node failures are common and stream processing systems must be able to recovery from such failures.

Castro Fernandez *et al.*, 2013 present ***recovery using state management*** (R+SM) and show that it can recover stateful operators more quickly and efficiently than other fault-tolerance approaches. In this project, R+SM is implemented in Storm and evaluated in comparison with Storm's existing ***source-replay*** fault-tolerance mechanism. R+SM is shown to be capable of recovering a failed operator more quickly than source-replay.

3. Contents

Contents

4. Introduction and objectives

Modern enterprises generate ever-growing volumes of data and big data analysis is now an effective business strategy. ***Stream processing systems*** (SPSs) are designed to process continuous streams of data in real-time. They have been the subject of a considerable body of work and have progressed from centralised systems, to parallel systems capable of being distributed across hundreds of servers. One such SPS that has seen widespread adoption in industry (*Powered By Storm*, no date) is ***Apache Storm*** (*apache/storm*, no date).

By making a large number of servers available as a utility, cloud computing could be well suited to accommodate these modern SPSs. However, given inevitable machine and network failures, SPSs must be able to recover efficiently.

Recovery using state management (R+SM) (Castro Fernandez *et al.*, 2013) is a fault-tolerance mechanism for SPSs where operator state is externalised and periodic state checkpoints are made by the SPS. Recovery from failure can be achieved by restoring backed up state to a standby node. R+SM has been shown to achieve faster recovery of stateful operators when compared to the existing approaches used by popular industrial SPSs such as Apache Storm.

The aim of this project is to determine whether R+SM can perform better than existing fault-tolerance strategies in an industrial stream processing system. This will be achieved by implementing R+SM in Storm – an SPS with significant commercial take-up. Implementation will be evaluated in comparison with Storm's existing ***source-replay*** (SR) fault-tolerance mechanism; to ascertain which approach can recover a failed node most quickly and with the lowest overhead.

Information is commercially valuable; an SPS can recover from failure the quicker information can be extracted from data. This project will make a contribution to the knowledge by evaluating whether R+SM is an applicable fault-tolerance solution in commercial Stream processing systems. This contribution will benefit the academic community researching SPSs by evaluating R+SM in a new context. It will also benefit the large community of Storm users and developers as the work could be extended to become an additional fault-tolerance strategy in Apache Storm.

I selected this challenging topic because it involved a number of interesting technologies in a cutting edge field of research. Clojure is a modern programming language that exposed me Lisps and functional programming. Distributed systems are an area of great interest to me and are very relevant in today's cloud orientated software industry and SPSs. Finally SPSs are an area of rapid development. New SPSs are being developed all the time, and there are many opportunities for employment for those with the necessary skills.

The project has the following objectives:

- Review existing research on fault-tolerance in SPSs.
  This objective will be deemed to have been met if a summary of the main techniques and works is provided in the context section of this report
- Design and implement R+SM in Apache Storm.
  This objective will be deemed to have been met up on the successful recovery of a failure in 3 operator query in Apache Storm.
- Design and perform experiments to collect performance data for R+SM vs SR approaches to recovery.
  This objective will be deemed to have been met if recovery time, latency and throughout data for recoveries using SR and R+SM are generated.
- Analyse and evaluate results and report on findings.
  This objective will be deemed to have been met if findings are provided in the results section of this report.

A combination of the ***design and creation*** and ***experiments*** (Briony J. Oates, 2006) research processes will be used to meet the objectives. The design and creation element will yield the implementation of R+SM and will be performed according to the ***prototyping*** (Dawson, 2005) software development process.

The structure of this report is as follows:

- **Chapter 5 – Context** sets the project in context by exploring the existing research and in the field of SPS fault-tolerance
- **Chapter 6 - Background** provides explanations R+SM and other concepts useful in understanding the chapters that follow
- **Chapter 7 – Methods** describes the research methods and software development practices applied. Methods specific to experiments are located in the chapter 9 with the results.
- **Chapter 8 – Design** beings with an overview of the Storm and its existing SR fault-tolerance approach. It goes on to cover the design of R+SM in Storm and finally to provide more in-depth discussion of several interesting aspects of the implementation. Design relating to the experiments is located within chapter 9.
- **Chapter 9 – Results** provides and overview of the experiments performed, followed by details of the results generated from the experiments and discussion of those results.
- **Chapter 10 – Discussion** covers the outcomes of each objective and explains how the objective was met. It also covers further work and findings.
- **Chapter 11 – Evaluation, reflections, conclusions** covers my thoughts on how successful the project was as a whole and how I would execute it differently if doing it again.

5. Context

Big data analysis is now an effective business strategy, which has led to the development of a multitude of processing frameworks (Agrawal, Das and El Abbadi, 2011, pp. 1–2). Complimenting batch processing systems such as MapReduce (Dean and Ghemawat, 2008), stream processing systems (SPS) provide real-time processing of continuous streams of data in a manner that can handle sudden increases in input volume (Balakrishnan *et al.*, 2004, p. 1).

SPSs have been the subject of considerable research (Abadi *et al.*, 2005, p. 1) progressing from centralised systems such as Aurora (Cherniack *et al.*, 2003) to distributed systems such as Borealis (Abadi *et al.*, 2005). The current crop of highly scalable SPSs such as SEEP (Castro Fernandez *et al.*, 2013), Apache S4 (Neumeyer *et al.*, 2010) and Apache Storm (*Storm, distributed and fault-tolerant realtime computation*, no date) are capable of a high level of parallelism and distribution (Castro Fernandez *et al.*, 2013, p. 1).

The large resource pools and on-demand model offered by cloud computing providers (Zhang, Cheng and Boutaba, 2010, p. 1) are well-suited for modern SPSs, however failures are common in cloud environments (Zhang, Cheng and Boutaba, 2010, p. 6) as such SPSs must be able to recover in an efficient manner (Castro Fernandez *et al.*, 2013, p. 1).

Various approaches to SPS fault-tolerance are presented in the literature. Approaches fall into two categories known as ***active*** and ***passive replication*** (Balazinska, Hwang and Shah, 2009). In active replication, a redundant secondary query is processed alongside the primary query, in the event of failure of a primary node, the system can failover to the secondary. The techniques within the passive replication category are ***passive standby*** and ***upstream backup***. In passive standby, the system regularly checkpoints operator state to backup nodes. To recover from failure the backup is first brought up-to-date by replaying tuples not present in the checkpoint; it can then replace the failed node – a process known as query repair. Upstream backup is similar to passive standby; however there is no checkpoint step. To recover from failure, the backup is brought up-to-date only by replaying tuples held in the output queues of upstream operators.

An additional approach to SPS recovery is the ***source-replay*** (SR) approach utilised by Apache Storm. This may be considered to be a special case of upstream backup. In Storm's implementation, the id of each tuple entering the query from the source, and any derived tuples, are tracked as a directed acyclic graph using a probabilistic hashing technique called Zobrist hashing (Zobrist, 1970). Inbound tuples are retained at the source and replayed if they or any tuples derived from them are not explicitly acknowledged within a specific period of time.

(Castro Fernandez *et al.*, 2013) consider a number of fault-tolerance strategies from the literature in their discussion of the recovery of stateful operators, which are facilitated in modern systems such as Apache S4 and Apache Storm. They argue that active replication is too resource intensive; that passive replication is less resource intensive but too slow to recover; and that upstream backup and SR are not practical for the recovery of operators that have state based upon the whole of the processed stream. With reference to Storm's SR approach, they also state that it ignores stateful operators (2013, p. 2) and that during recovery there is a negative effect on performance (2013, p. 3).

The nature of SR does support the claim that performance could be reduced during recovery, this is because in order to remain recoverable, a stateful bolt would be unable to mark as processed any tuple upon which its state depends. In the event of failure every source tuple would be need to be replayed. Beyond the work of Castro Fernandez et al, it has not been possible to locate attention to the SR recovery approach within the academic literature.

(Castro Fernandez *et al.*, 2013) contribute R+SM. This approach defines a model for representing ***query state***, consisting of the ***processing***, ***output buffer*** and ***routing*** states of all operators in the query. Also defined are a set of operations for managing state and using it to recover from failures. To recover from an operator failure, the SPS can use the latest operator ***checkpoint*** to backup and restore operator state to a node from a pool of standby nodes. The node can then be brought up-to-date using the output buffer state of upstream operators.

They observe that for a two operator, stateful query their approach yields faster recovery times than the upstream backup and SR approaches. They also evaluate the overhead of their approach by exploring processing latency, but not in comparison to other approaches. Such comparison would have been useful; especially since the latency of Storm's SR approach has not been explored in the literature.

6. Background

This chapter provides an overview of concepts later referred to in this report. It begins with coverage of R+SM as described by Castro Fernandez *et al.* and goes on describe the decorator pattern which is referred to in the design chapter.

6.1. R+SM overview

In their definition of R+SM (Castro Fernandez *et al.*, 2013) describe a model for exposing operator state to the SPS and operations for acting upon that state. The model and operations can be used both for operator scale out and to recover operators in the case of failure; this project is concerned only with failure recovery.

Three types of operator state are defined: ***processing state***, ***buffer state*** and ***routing state***.

Processing state represents is the state held by a stateful operator, it is a product of the processing of inbound tuples, possibly in the form of a summary, and is likely to be used in the creation of outbound tuples. For example a word count operator takes as input a stream of word tuples, a count is maintained for each word seen and these counts will periodically be emitted as output tuples. In this example the list of words, and associated counts held by the operator comprise its processing state. The model for processing state is formed of a map of partition keys to associated state, and an ordered collection of timestamps. The timestamps indicate the latest timestamp of tuples from which the state associated with each key is derived.

In the upstream backup approach to SPS fault-tolerance, operators keep a copy of any tuples that they emit, so that they can be replayed in case of failure to rebuild the processing state of downstream operators. R+SM also uses this technique, and refers to these stored tuples as buffer state. Unlike upstream backup, there isn't a requirement to store the entire of history of tuples reflected in the processing state of downstream operators. Only those tuples not reflected in a checkpoint need to be retained. Buffer state is modelled as a collection of tuples and associated timestamps per downstream operator partition. The operator is able to remove from the buffer state all tuples with timestamps preceding a given timestamp. This is necessary when a downstream operator performs a checkpoint and the buffered tuples are no longer required for recovery. For example, a sentence split operator takes a sentence and splits it into words which are emitted as separate tuples to a partitioned downstream word count operator. The split sentence operator will keep a collection of tuples with timestamps for each downstream operator partition that it emits tuples to. When a word count operator performs a checkpoint it sends the timestamp of the latest tuple represented in the checkpoint to the split sentence operator, any buffered tuples for that partition that have timestamps older than the checkpoint timestamp will then be removed from the buffer state.

If an SPS allows operators to be partitioned at runtime then the downstream operators that an operator sends tuples to may also change during the lifetime of the query. Again considering the split sentence operator described above; the target downstream operator for a given word tuple might be determined by the first letter of the word which is used as a ***partition key***. If the word count operator is partitioned (say from two partitions to three) then the target downstream operator for a given word tuple may change. This association of partition keys (in this case the first letter of the word) with specific partitions of a downstream operator is referred to as routing state with R+SM. Storm has a concept called *grouping* which is synonymous with the aforementioned routing. Storm groupings are fixed at design time and cannot change within the lifetime of a query. Therefore routing state is not in scope for this project.

Four operations are defined for the management of state:

1. ***Checkpoint***
2. ***Backup***
3. ***Restore***
4. ***Partition***

The partition state operation relates to dynamic operator scale-out and is outside the scope of this project.

The checkpoint operation requests the processing state of an operator. The operator upon which the checkpoint is operating returns its processing state in the format described above.

Processing state retrieved from an operator by the checkpoint operation is backed up as part of the backup operation. Here the processing state is stored on an upstream operator and the timestamp of the latest tuple in the processing state can be used to trim the buffer state of upstream operators.

After failure, an operator partition will be re-created by the SPS. The restore operation retrieves processing state from its backup location and restores it to the recovering operator. It then replays tuples held in the buffer state of upstream operators and uses the checkpoint timestamp to discard any replayed tuples which have timestamps earlier than processing state creation time. These tuples are already reflected in the checkpoint and should therefore not be reprocessed by the recovering operator. The tuples that are replayed to the recovering operator will be those tuples that were emitted from upstream operators since the last checkpoint, as any state that was based on them will have been lost during the failure.

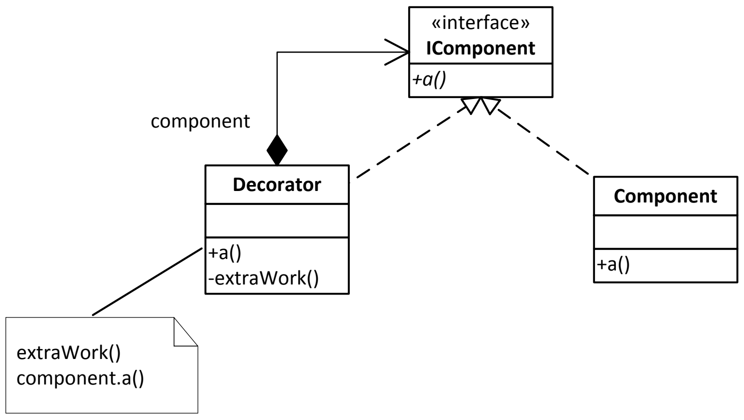
6.2. The decorator pattern

This section describes the decorator pattern which is used in the design of R+SM for Apache Storm and referred to in design chapter of this report.

The ***decorator*** pattern (Gamma *et al.*, 1994, p. 175) is categorised as a structural design pattern in that it deals with how objects can be composed to form a system. It facilitates the adding of functionality, or roles, to an object without the modification of that object. Multiple decorators can be composed with an object to assign multiple roles to that object. This approach does not require changes to the original class so is a flexible way of mixing and matching functionality to objects and avoiding the proliferation of multiple base classes that provide every combination of roles. The approach also helps to maintain separation concerns as each decorator can have a single role, making them easier to test.

The decorator conforms the same interface and holds a reference to the decorated object – the ***component***. The decorator collaborates with other objects in place of the component; when its methods are called it performs its work and then calls the component.

The following UML class diagram shows a simplified decorator.



7. Methods

This chapter discusses the research methods that I used to achieve the objectives outlined section 4 above. It begins by discussing the methods used for design and implementation and then goes on to discuss the methods used for experimentation.

The project objectives include the design and development of R+SM for Apache Storm, and the evaluation of R+SM in comparison with SR.

The design and development elements of the project use the ***design and creation*** (Briony J. Oates, 2006, p. 108) research process. The description of R+SM detailed in Castro Fernandez *et al.*, 2013 formed the requirements for the design, which is detailed in section 5.1 of this report. UML was used where appropriate for report diagrams. However UML was not capable of representing of all the ideas that I wished to show so for some diagrams I used my own informal notation.

The design and implementation was carried out according to the ***prototyping*** (Dawson, 2005, p. 124) ***software development lifecycle model*** (SLDC) (Dawson, 2005). Whilst the requirements for the project were well defined the difficulty of the design and implementation task was high. I highlighted implementation difficulty as a risk in the risk register that I submitted with the proposal (see appendix A). I began the process with a phase of ***throw-away prototyping*** in order to investigate the feasibility of the design (Dawson, 2005, p. 31). The results of this phase are described in section 8.4.3.1. I followed this initial phase with two phases of ***evolutionary prototyping*** (Dawson, 2005, p. 126), the results of which can be seen as the two phases in the results section (9.2 and 9.3) of this report.

The phases described above are consistent with the project plan supplied with the proposal though the experimental evaluation section was performed alongside the design/development phases, as well as in its scheduled slot.

The development took place ***top-down*** (Dawson, 2005, p. 134)***.*** Once the implementation level had been selected the main classes were designed (detailed in section 5.4.1) with the design and implementation being carried out at progressively lower levels. This ensured that the design fitted with Storm. By contrast if I had started by designing and developing the detail and working bottom up towards integration with Storm, there would have been no guarantee that the design would integrate correctly.

I used the ***JetBrains IntelliJ*** (*IntelliJ IDEA — The Most Intelligent Java IDE*, no date) IDE running on Ubuntu LInux 14.04 LTS (*The leading OS for PC, tablet, phone and cloud | Ubuntu*, no date) for software development. When running a Strom cluster locally I ran the cluster on ***Docker*** containers (*Docker - Build, Ship, and Run Any App, Anywhere*, no date) using ***Storm Docker*** (*wurstmeister/storm-docker*, no date) and ***fig*** (*docker/fig*, no date)**.**

Verification of the design and implementation took place through unit testing and by the extensive exercising that the solution got from running the experiments. I unit tested the main classes in isolation of the other classes to ensure that they behaved as designed. I was able to isolation the class under test using ***mocks*** (Mackinnon, Freeman and Craig, 2001) and the ***Mockito*** (*mockito/mockito*, no date) mocking library. It was not possible to write an automated test to perform a true operator failure and recovery since tests run with Storm running in local mode, where all daemons run in a single process. Throwing an exception to cause an operator failure would fail the entire test.

Verification was also achieved by writing entries to the log files throughout my implementation. When performing the experiments detailed in chapter 9 I verified that the software was working correctly according to the requirements by regularly checking the log files to ensure that the system was running correctly.

Validation that the requirements themselves are correct and provide benefit is the subject of the project objectives "Design and perform experiments to collect performance data for R+SM vs SR approaches to recovery" and "Analyse and evaluate results and report on findings" as listed in chapter 4. These objectives are concerned with the validation or R+SM as an applicable and preferable technique for fault-tolerance in SPSs.

R+SM will be validated by implementing it in Storm, and then by performing experiments of its performance in comparison with SR, Storm's existing fault-tolerance approach that is already used widely in industry.

"Design and perform experiments to collect performance data for R+SM vs SR approaches to recovery" and "Analyse and evaluate results and report on findings" will use the ***experiments*** (Briony J. Oates, 2006, p. 126) research process.

The hypothesis to be tested is:

The R+SM recovery mechanism can be efficiently integrated in the Apache Storm SPS.

Discussion of experiment design, measures of efficiency, dependant and independent variables and methods used to carry out the experiments are provided in chapter 9.

8. Design

This chapter beings with an overview of the Storm SPS, its general concepts and programming model. This is followed by an overview of the elements of Storm's architecture that are relevant to the R+SM design; then an overview of Storm's existing SR fault-tolerance mechanism.

The chapter then goes on to detail the design of R+SM for Storm before covering several interesting aspects of the implementation in more detail.

8.1. Storm overview

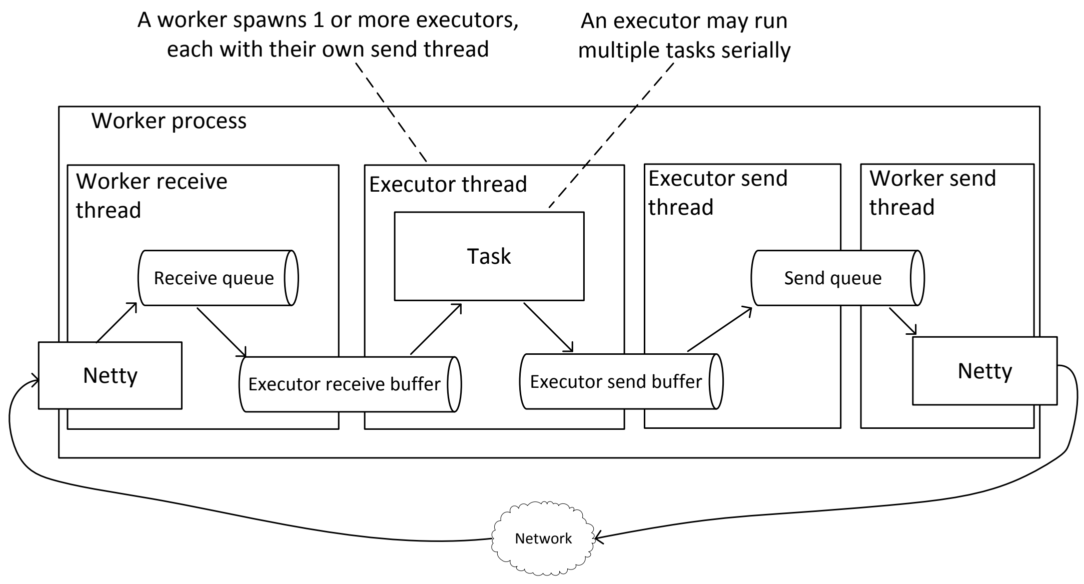
Storm is an Apache project that was originally developed by the company Backtype before they were acquired by Twitter who open-sourced it. This SPS was selected for implementation of R+SM due to its active user base (*Powered By Storm*, no date) and open-source community. The source code for Storm is freely available under the Apache License on GitHub (*apache/storm*, no date); including a comprehensive suite of tests. Implementation logic and tests are written in Clojure, a functional language for the JVM in the Lisp family of languages. All interfaces are written in Java and the project uses the Apache Thrift (*Apache Thrift - Home*, no date) RPC framework to allow client usage by a wide range of languages.

The system aims to provide a flexible programming model, applicable to a range of problems and to hide much of the incidental complexity associated with distributed, parallel and fault-tolerant stream processing; allowing users to concentrate instead on business logic. These features have made Storm a very popular system (*Powered By Storm*, no date) with many installations.

Computations in Storm are known as *topologies* and are represented as a graph of user-defined operators (known as *components* in Storm), that can be one of two types - *spouts* and *bolts*. Spouts output one or more continuous streams of data items known as *tuples*, resulting in topologies being continuous – that is they only end when killed by the user. Bolts take one or more streams of tuples as input and emit one or more streams of tuples as output; they may create new tuples and can contain arbitrary state.

A runtime instance of a component and is called a *task* and can be considered to be a partition of a partitioned operator. Tasks are executed by an *executor*. Executors are implemented as threads and are spawned by a *worker*. Workers are implemented as processes. Since one or more tasks can run per component, this architecture allows work to be parallelised across multiple threads and processes on a node or across nodes.

The following diagram shows the different processes, threads and queues that are used by Storm:

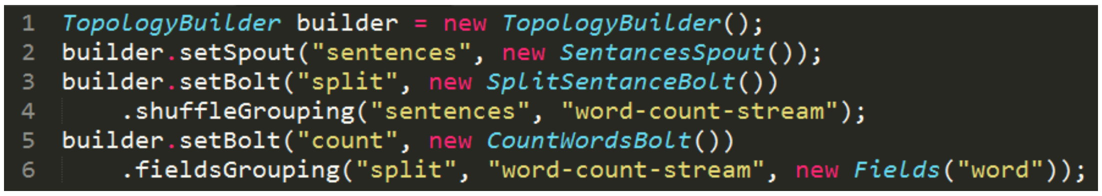


Groupings specify how streams of tuples are routed between tasks, Storm includes a number of built in groupings such as *fields*, which routes tuples based upon one or more fields in the tuple payload, *direct*, which routes tuples to specific tasks and *shuffle*, which distributes tuples evenly across downstream tasks. Groupings may also be user-defined.

Both the stream groupings, and the number of tasks to run per component are fixed at design-time, however the number of workers and executors can be changed and the topology can be rebalanced to distribute tasks evenly across the available resources.

8.1.1. How a topology is created and executed

Topologies are created using Storm's `TopologyBuilder` class, which provides methods to add components to a topology, link them together by applying stream groupings and to specify the level of parallelism of each bolt. For example a simple word count topology with three linked components might be defined as in the following snippet:



```
1  TopologyBuilder builder = new TopologyBuilder();
2  builder.setSpout("sentences", new SentancesSpout());
3  builder.setBolt("split", new SplitSentanceBolt())
4      .shuffleGrouping("sentences", "word-count-stream");
5  builder.setBolt("count", new CountWordsBolt())
6      .fieldsGrouping("split", "word-count-stream", new Fields("word"));
```

Line 1: Creates the topology.
Line 2: Adds a spout of type `SentencesSpout` with the id *sentences*.
Line 3: Adds a bolt of type `SplitSentenceBolt` with the id *split*.

Line 4: Sets the input of the split bolt to be the tuples on stream *word-count-stream* that are output from the sentences bolt. The split task that receives a given tuple will be selected at random.
Line 5: Adds a bolt of type `CountWordsBolt` with the id *count*
Line 6: Sets the input of the count bolt to be the tuples on *stream word-count-stream* are output by the split bolt. The "count" task that receives a given tuple will be selected based on the "word" field of the tuple - the same count task will receive all tuples for a given word.

The above code snippet does not show how a component declares the streams and fields of the tuples that it emits. This takes place within the component class itself and is discussed below.

The key methods of a spout component are:

- **`open(java.util.Map conf, TopologyContext context, SpoutOutputCollector collector)`** – called by Storm to tell the spout to prepare itself to emit tuples when nextTuple is called. The collector parameter that is passed to this method is typically assigned to a member variable as it is required in order to emit tuples from the nextTuple method.
- **`nextTuple()`** – Storm repeatedly calls this method to request that the spout emits another tuple into the topology. This approach means that Storm pulls tuples into the topology, rather than them being pushed by the spout. The spout typically emits one tuple per call to nextTuple but may emit multiple tuples if required. Tuples are emitted by calling the emit method of the collector member variable that is passed to the open method.
- **`declareOutputFields(OutputFieldsDeclarer declarer)`** – this method is used by the spout to declare what fields the tuples that it emits will have, and what streams the tuples will be emitted into.

The key methods of a bolt are:

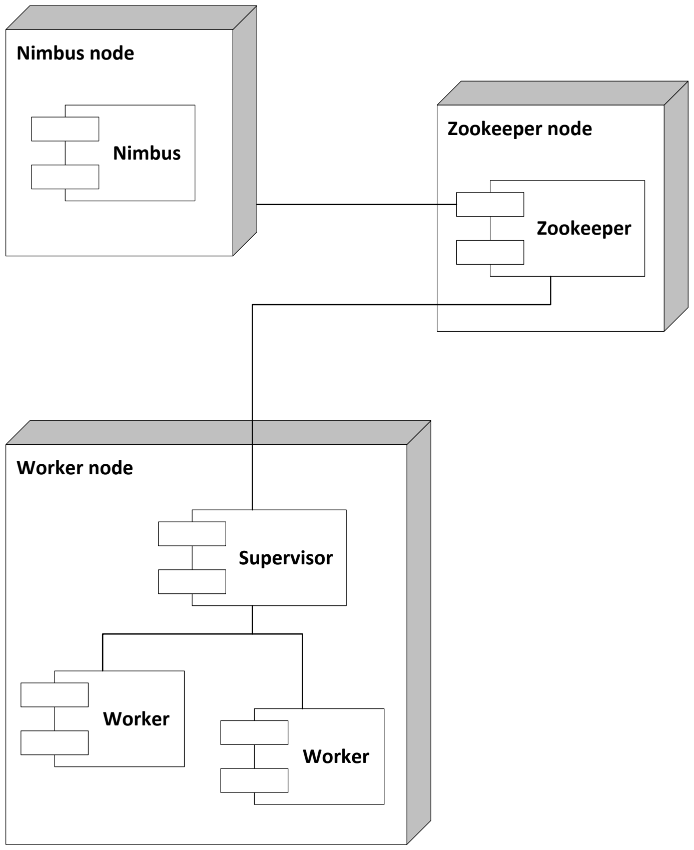
- **`prepare(java.util.Map conf, TopologyContext context, OutputCollector collector)`** – called by Storm to tell the bolt to prepare itself to process tuples when it's execute method is called. The collector parameter that is passed to this method is typically assigned to a member variable as it is required in order to emit tuples from the emit method.
- **`execute(Tuple input)`** – when a tuple is sent to a task, it is passed as a parameter to the execute method of the task. The task will often, but not necessarily create new tuples and send them downstream by calling the emit method of the collector member variable that was passed to the prepare method
- **`declareOutputFields(OutputFieldsDeclarer declarer)`** – same as spout and documented above.

Notice that spouts have an open method, and bolts have a prepare method that are used in a similar way to a how a Java class constructor would be used – to provide the component with its context/dependencies and prepare it start doing its work. This is necessary because the classes are instantiated when they are passed to the topology builder, and this takes places on the user's local machine. Storm serializes the instantiated objects and de-serializes them on each worker node where they need to run. Since de-serializing a class does not call its constructor, another mechanism by which to provide its runtime context is necessary.

8.2. Storm architecture

Storm is designed to run distributed across a cluster of machines. There are three types of nodes in a cluster, these are Apache Zookeeper (Hunt *et al.*, 2010) nodes, a *Nimbus* node and one or more worker nodes (*Apache Storm*, no date). The Nimbus node is the cluster master and is responsible for receiving topologies submitted by the user, starting worker nodes, distributing work and monitoring. Nimbus communicates with workers via Zookeeper. Each worker runs a supervisor daemon which starters and stops worker processes as directed by Nimbus. In addition a Storm UI service runs as a separate service to host a website providing information about the cluster and any running topologies.

The following UML component diagram shows the main elements of the Storm architecture:



The Nimbus, Supervisor and Worker processes are implemented as daemons and are written in Clojure. In the event of exceptions, the daemons are designed to fail-fast and be restarted. The cluster can continue to processing its workload whilst any of the nodes are unavailable, and these nodes will re-join the cluster when they become available again.

Transport between nodes is handled by the Netty (*Netty: Home*, no date) network IO framework. A worker process has a send thread and a receive thread, each with their own queue to prevent overload. All executors run by a worker share the same send and receive threads. Each executor has its own inbound and outbound buffers to regulate the flow of tuples in and out, and prevent overloading. A dedicated thread to take tuples from the output buffer and place them on the Netty transfer queue. The worker transfer-out queue and the executor inbound and outbound buffers use the LMAX Disruptor (Thompson *et al.*, 2011) library.

There are two modes of operation in Storm, these are local mode and distributed mode. Local mode is used for development and testing and runs the entire cluster in a single process so that it can be executed from a developer's local machine or IDE. In distributed mode each daemon runs as its own process, this mode is used for production deployments.

Topologies are submitted to the Nimbus daemon using a Storm executable. This executable must be installed locally on the machine that is initiating the submission, it instantiates the topology and serializes it to send to Nimbus, which then distributes the serialized classes to Supervisor daemons on the worker nodes.

The configuration API also allows for each component to request a *tick tuple* at specific intervals, for example once per second. This approach simplifies scheduling of actions at intervals, which would normally require the use of timers or additional threads.

There are a multitude of configuration options for Storm, controlling many aspects of its operation, including the lengths of any timeouts, buffer sizes, the number of executors per worker and whether Storm's SR fault-tolerance is active. These configuration options may apply at different levels of the cluster – for example they may affect the entire cluster, specific daemons, specific topologies or specific components.

The code base consists of three main layers - an Apache Thrift layer, a Java API and an implementation layer. The Apache Thrift exposes Storm's primary abstractions as Thrift interfaces so that they can be consumed from any language that has an Apache Thrift client. The main classes and interfaces are written in Java, however the implementation logic is predominantly written in Clojure (*Structure of the Codebase*, no date). The Clojure language has a strong focus on concurrent programming (Kraus and Kestler, 2009) that is well suited to a framework such as Storm that uses multiple concurrent threads.

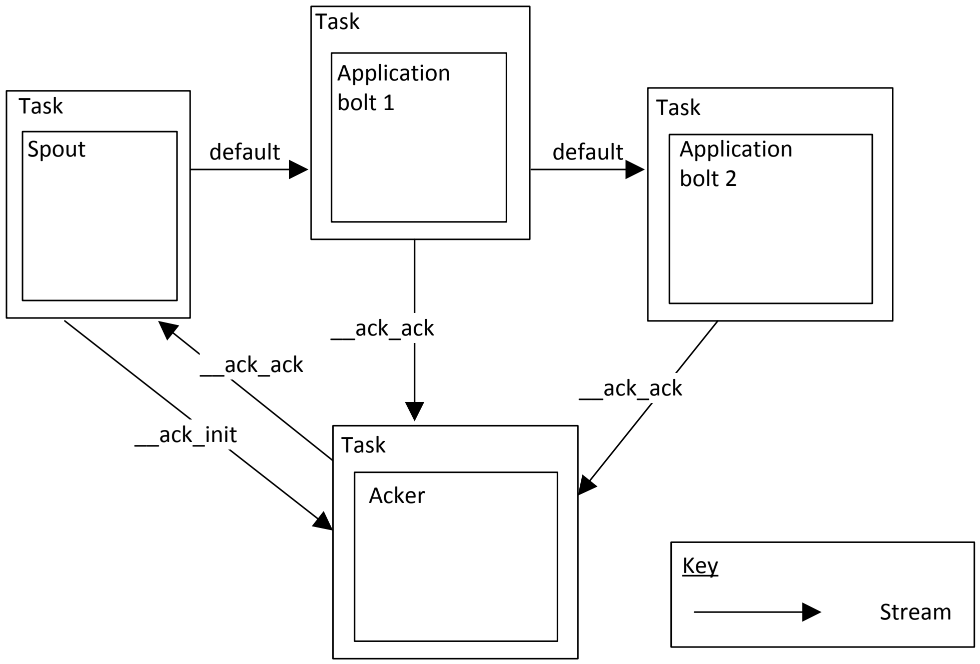
Storm uses Maven as its build and dependency management tool. Building changes to the source code involves first running Maven lifecycles to validate, compile, test, package and install (to the local Maven repository) the entire project and then run a separate set of lifecycles to produce distribution binaries which can be run as a cluster in distributed mode.

8.3. Fault tolerance in Storm

When a tuple emitted from a spout (a spout tuple) is processed by a bolt, that bolt may emit one or more new tuples, which when processed by downstream bolts, may result in further tuples being created;

resulting in a directed acyclic graph of tuples. Fault-tolerance in Storm uses the SR model, which guarantees that every spout tuple will be processed. The user is required to inform the system whenever a new tuple is created and when a tuple is processed; the system's 'Acker' service uses this information to track the complete processing of tuple graphs, replaying the tuple from the spout if the tuple graph is not marked as complete within a customisable time period – 30 seconds by default. The Acker service is implemented as a bolt and communication between the Acker and application bolts takes place by sending tuples on hidden streams called *__ack_init* and *__ack_ack*. The Acker uses a probabilistic hashing technique called Zobrist hashing (Zobrist, 1970) to monitor when tuples have been processed. In this technique a hash (known as an 'ack val') is associated with the ID of the spout tuple, this hash contains an XOR of the ids of all of the unprocessed tuples still in the graph. When the ack val reaches 0, the system knows the entire graph is processed and removes the spout tuple from the source.

The following diagram shows the components and streams that form Storm's SR fault-tolerance functionality:



One limitation of this approach is that for a stateful operator to be recoverable, it cannot acknowledge any tuple upon which its state depends – tuples are removed from the source when acknowledged. Since tuples are automatically marked as failed if they are not acknowledged within 30 seconds (by default) of creation, an operator can only recover state based on a 30 second window of tuples.

8.4. R+SM design and implementation

In this section I describe my design for R+SM in Apache Storm. I begin by describing how my design fits into Storm's architecture then discuss the main components of my contribution. I then show how the components can be used to build a recoverable topology before describing how the solution performs normal processing and recovery.

I also provide a list of the main classes in the design before discussing several interesting implementation details in more depth.

8.4.1.Design

I implemented R+SM at the Java layer of the Storm. Being designed for users, this level is less complex than the Clojure implementation layer and provides a clean set of abstractions to programme against, it is also documented, which the Clojure layer is not. Further, changes at this layer can be deployed to Storm as part of a topology and so do not require a distributable release build of the entire framework in order to run. As a result of the reduced time taken to compile and test code, the feedback loop is thus also reduced. Given the above, design and implementation at the Java layer was deemed to be the most appropriate choice to meet the project’s objectives.

In designing how R+SM could be applied in Storm, two main roles of functionality were identified. The areas of functionality are:

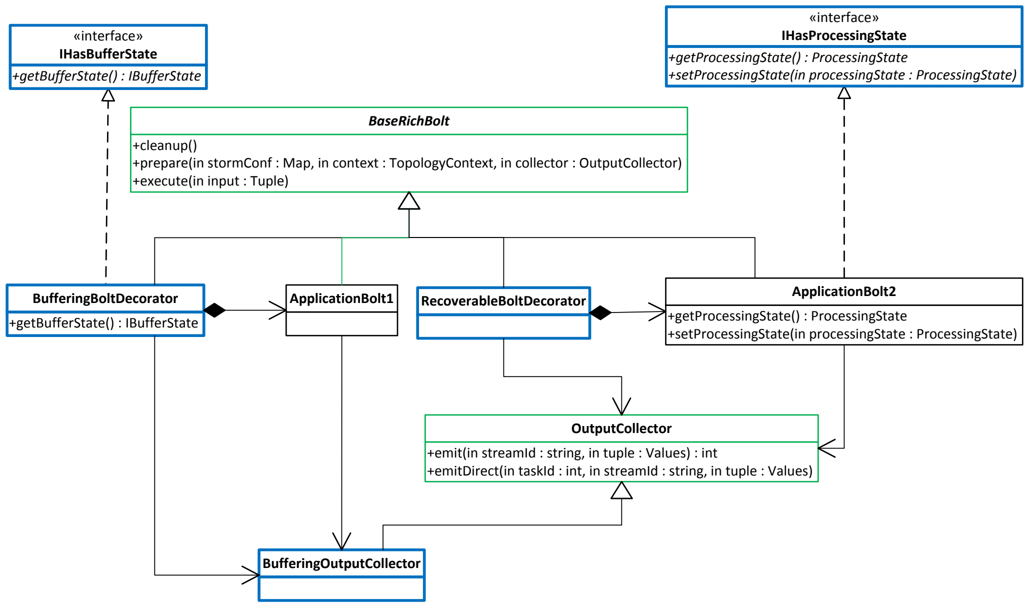
1. The ability of a task to act as an upstream backup for another task
2. The ability of a task to be recoverable by performing checkpoints

In addition the system should be able to recover a failed task by restoring processing state from a checkpoint and replaying tuples from the upstream backup.

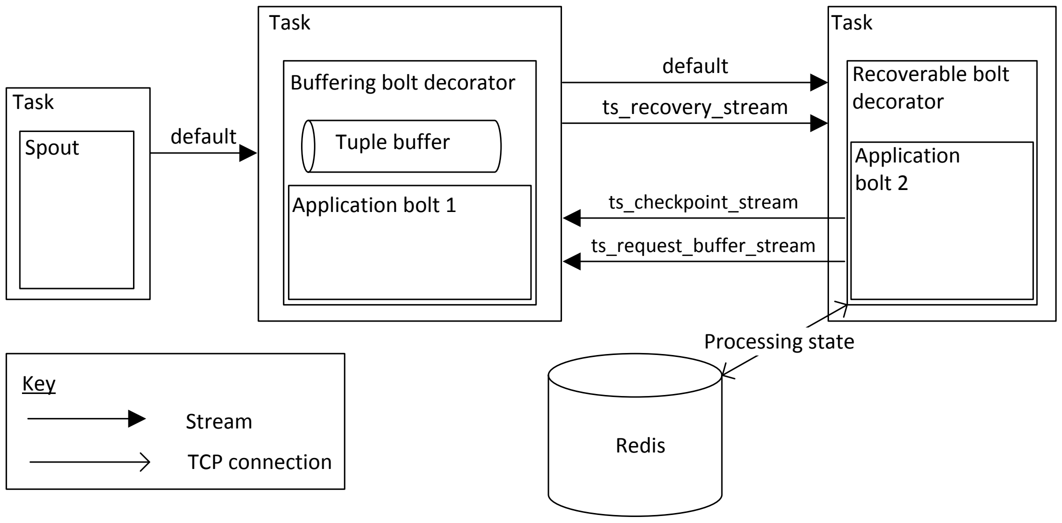
The main classes in the design are `BufferingBoltDecorator` (***BBD***) and `RecoverableBoltDecorator` (***RBD***), which both apply the decorator pattern (see section 6.2). BBD performs the role of making a task act as an upstream backup for another task. RBD performs the role of making a task recoverable by performing checkpoints.

One or both of these classes can be composed with a bolt, to make the bolt act as an upstream backup, be recoverable or both. This approach maintains a separation of concerns between the two roles, and between the fault-tolerance and application functionality.

The following diagram shows the main classes in the R+SM design for Storm. Classes highlighted in green are part of Storm, classes highlighted in blue are added as part of the design:


«interface»
IHasBufferState
+getBufferState() : IBufferState
«interface»
IHasProcessingState
+getProcessingState() : ProcessingState
+setProcessingState(in processingState : ProcessingState)
BaseRichBolt
+cleanup()
+prepare(in stormConf : Map, in context : TopologyContext, in collector : OutputCollector)
+execute(in input : Tuple)
BufferingBoltDecorator
+getBufferState() : IBufferState
ApplicationBolt1
RecoverableBoltDecorator
ApplicationBolt2
+getProcessingState() : ProcessingState
+setProcessingState(in processingState : ProcessingState)
OutputCollector
+emit(in streamId : string, in tuple : Values) : int
+emitDirect(in taskId : int, in streamId : string, in tuple : Values)
BufferingOutputCollector

The following diagram shows the main components and streams in the R+SM design for Storm:

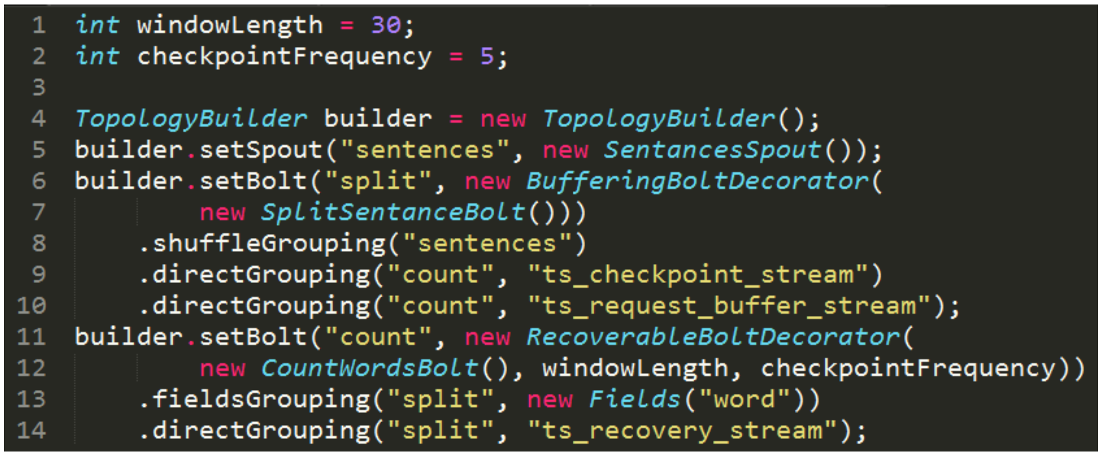


The design uses tuples as a means of communication between bolts. This takes advantage of Storm's existing infrastructure for transporting and routing messages rather than adding a separate mechanism. Streams are used to identify tuples that have different purposes within the fault-tolerance approach and to keep these tuples separate from the application tuples.

The following Streams are used and need to be declared at topology creation time:

Stream	Description
default	The default stream for applciation tuples
ts_request_buffer_stream	Stream for recovery request tuples
ts_recovery_stream	Stream for replayed tuples
ts_checkpoint_stream	Stream for checkpoint notification tuples from a recoverable bolt to an upstream backup

The following example shows the creation of a word counting topology with a recoverable count bolt. The `TopologyBuilder` and `Fields` classes in the example are part of Storm. The `BufferingBoltDecorator` and `RecoverableBoltDecorator` are part of my design, and the `SentancesSpout`, `SplitSetanceBolt` and `CountWordsBolt` classes are provided for illustrative purposes.



```
 1  int windowLength = 30;
 2  int checkpointFrequency = 5;
 3
 4  TopologyBuilder builder = new TopologyBuilder();
 5  builder.setSpout("sentences", new SentancesSpout());
 6  builder.setBolt("split", new BufferingBoltDecorator(
 7          new SplitSentanceBolt()))
 8      .shuffleGrouping("sentences")
 9      .directGrouping("count", "ts_checkpoint_stream")
10      .directGrouping("count", "ts_request_buffer_stream");
11  builder.setBolt("count", new RecoverableBoltDecorator(
12          new CountWordsBolt(), windowLength, checkpointFrequency))
13      .fieldsGrouping("split", new Fields("word"))
14      .directGrouping("split", "ts_recovery_stream");
```

Line 4: Creates a topology builder object.
Line 5: Creates spout called *sentences*
Line 6: Creates a bolt called *split*, and passes in a new buffering bolt decorator so that the bolt can act as an upstream backup
Line 7: Passes a split sentences bolt to the buffering bolt decorator. This is the application bolt being decorated to add the upstream backup functionality
Line 8: Links the input of the split sentences bolt to the output of the sentences spout on the default stream. Tuples on this stream are the application tuples.
Line 9: Links the split bolt to receive tuples from the count bolt on stream *ts_checkpoint_stream* using a direct grouping, these tuples will indicate when to trim the buffer of the upstream backup. The direct grouping allows the split task emitting a tuple to indicate which task to send the tuple to.
Line 10: Links the split bolt to receive tuples from the count bolt on stream *ts_request_buffer_state_stream* using a direct grouping. These tuples will request that tuples be replayed in order to recovery a failed task.
Line 11: Creates a bolt called *count* and passes in a new recoverable bolt decorator so that the bolt is recoverable.

Line 12: Passes a count words bolt to the recoverable bolt decorator. This is the application bolt being decorated to make it recoverable. The window length and checkpoint frequency (both in seconds) are also passed in.
Line 13: Links the count bolt to receive tuples from the split bolt on the default stream – these are the application tuples.
Line 14: Links the count bolt to receive replayed tuples from the split bolt as part of a recovery.

The BBD composes with an application bolt to allow it to act as an upstream backup for downstream tasks. It is responsible for maintaining a buffer of any tuples that are emitted by the application bolt, replaying those tuples to recover a failed downstream task, and trimming old tuples from its buffer according to a given timestamp associated with a checkpoint of a downstream task.

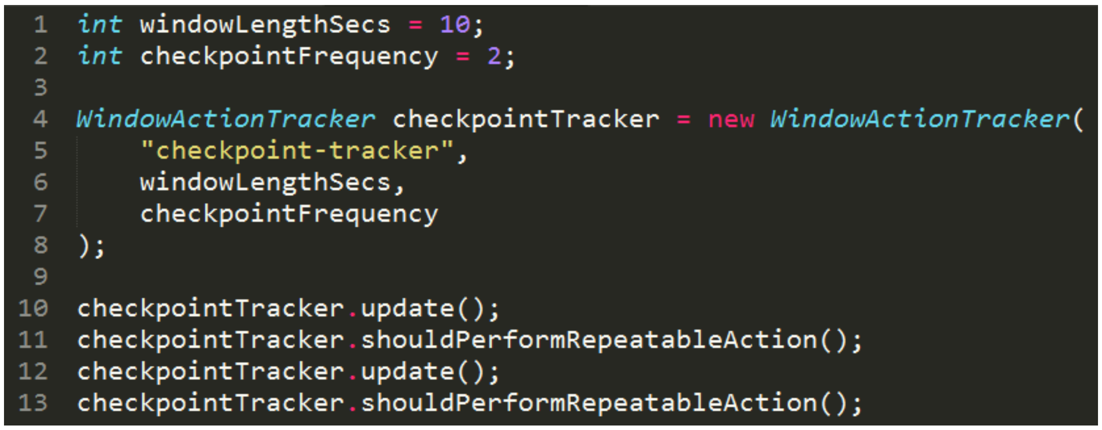
Tuple buffering is achieved using the `BufferingOutputCollector` (*BOC*) class, which is a subclass of Storm's `OutputCollector` class. When a topology is created the topology builder is passed a reference to an instantiated BBD, which in turn holds a reference to the instantiated application bolt (see UML class diagram above). When Storm calls the `prepare` method on the BBD, passing in an output collector, the BBD wraps the output collector in a new BOC, then calls `prepare` on the application bolt, passing in the BOC instance to use instead of the standard output collector. Every time the application bolt calls `outputCollector.emit(…)` the tuple is passed to the standard output collector, which returns the id of the task to which the tuple was routed. The tuple is then stored according to the downstream task id, and the current timestamp, facilitating efficient trimming of the appropriate buffer when a downstream task performs a checkpoint.

The BBD expects to receive tuples on the acknowledgement stream when a downstream operator performs a checkpoint. The payload of these tuples contains the timestamp of the most recent tuple that originated from the current (receiving) task that is reflected in the checkpoint. When the BBD receives a tuple on the acknowledgement stream, it finds the id of the task that sent the tuple (available by calling its `getSourceTask` method); any tuples buffered for this task, that have timestamps preceding the payload timestamp are removed from the buffer.

The RBD composes with an application bolt to make it recoverable by performing checkpoints at configurable intervals. Since an application bolt may hold state in any structure, the responsibility of converting the state into a general format that can be used for R+SM is assigned to the application bolt. The bolt implements interface `IHasProcessingState` which has methods for retrieving and restoring processing state, which has type `ProcessingState`. `ProcessingState` has a timestamp (the timestamp of the most recent tuple represented in the checkpoint) and the state itself as members. Considering the design in relation the R+SM as defined by Castro Fernandez *et al.* interface `IHasProcessingState` is the mechanism by which the bolt exposes it's processing state to the SPS.

To perform a checkpoint RBD retrieves the processing state from the application bolt, serializes it as a string and stores it in an instance of the ***Redis*** (*Redis*, no date) key value store, using the task id as the key, and the serialized processing state as the value. Next RBD notifies all upstream tasks that a checkpoint has been performed by sending them a tuple on the acknowledgement stream. For each upstream task, the tuple payload is the timestamp of the most recent tuple that originated at that task, which is reflected in the processing state that was part of the checkpoint.

The frequency at which to perform checkpoints is configured by passing the window length and the desired number of seconds between checkpoints to RBD's constructor. RBD receives Storm tick tuples once per second (configured in the `getComponentConfiguration` method). It uses tick tuples in combination with the `WindowActionTracker` class to monitor how far through a window it is at any time. Each time it receives a tick tuple it updates `WindowActionTracker` then asks it whether it should be perform a checkpoint. `WindowActionTracker` was inspired by the `SlidingWindowCounter` and `SlotBasedCounter` classes that are supplied with Storm.



```
 1  int windowLengthSecs = 10;
 2  int checkpointFrequency = 2;
 3
 4  WindowActionTracker checkpointTracker = new WindowActionTracker(
 5      "checkpoint-tracker",
 6      windowLengthSecs,
 7      checkpointFrequency
 8  );
 9
10  checkpointTracker.update();
11  checkpointTracker.shouldPerformRepeatableAction();
12  checkpointTracker.update();
13  checkpointTracker.shouldPerformRepeatableAction();
```

Line 15: Creates a `WindowActionTracker` object.
Line 16: Names the tracker *checkpoint-tracker.*
Line 17: Sets the number of slots in the tracker.
Line 18: Sets how often within the window the action should be performed.
Line 9: Moves the slot forward – now at 1
Line 10: Returns false
Line 11: Moves the slot forward – now at 2
Line 12: Returns true

The `shouldPeformRepeatableAction` method returns true or false depending on whether the action should be carried out in the current slot. In the example above, where the checkpoint frequency is set to 2 seconds, `shouldPeformRepeatableAction` will return true at slots 2, 4, 6, 8 and 10.

Identifying that a task has failed and re-instantiating it, are outside the scope of the project and are handled by Storm. Both the local Supervisor daemon and the Nimbus daemon monitor tasks so that they are aware when they fail. A failed task will be re-created by Storm on an available worker; the task id will be the same as it was before failure. When the task has been re-created it's prepare method will be called and processing can continue. When prepare is called on a RBD task, the task will attempt to recover itself in case it is being recreated after a failure. First it will check the Redis store to see if any checkpoints are available for its task id. If a checkpoint is found then the task is recovering from a failure, the RBD will deserialize the processing state and restore it back to the application task. RBD then requests that upstream backups replay any buffered tuples. This is achieved by sending a tuple on the recovery request stream to each upstream backup.

Upstream backup tasks (decorated by BBD) will be holding buffers of all tuples that were sent to downstream tasks since they last performed checkpoints. When an upstream backup receives a tuple on the recovery request stream it will find the id of the task that sent the tuple and replay any buffered tuples for that task on the recovery stream. When the recovering task (decorated by RBD) receives tuples on the recovery stream, it will compare their timestamp against the timestamp that was associated with the restored checkpoint. If the received tuple has a timestamp that is before the restored checkpoint timestamp then the tuple is already represented in the restored processing state and can be discarded, otherwise the tuple is reprocessed, bringing the task back to its pre-failure state so that normal processing can be resumed.

R+SM and this design require that each tuple has a timestamp indicating when the tuple was created. However in Storm this is not part of the tuple data structure at the Thrift, Java or Clojure layers. To support the required functionality without changing the framework itself, the tuple timestamp is added to the tuple payload. To achieve this `BufferingOutputCollector` adds the timestamp to the tuple payload after the tuple is emitted by the application bolt. With the current design, downstream bolts have to be aware of the added tuple timestamp.

Since the design treats each task (rather than bolt) independently with its own upstream buffer and processing state, it is able to recover partitioned operators and support all of Storms grouping types.

Storm does not support runtime partitioning of operators – the number of tasks created for a given bolt, and the associated stream groupings are fixed at design time; runtime scaling is achieved by varying the number of available workers and executors. When using the shuffle grouping the downstream operator to which a given tuple will be routed is not known ahead of tuple creation however tuples are buffered according to the id of the task that they are sent to and replayed to the same task when recovering. As such this design does not address explicit backup or recovery of routing state as this is only required when operators are partitioned dynamically at runtime.

Topologies that use this R+SM approach to not require Storm's source-replay fault-tolerance approach to be available at the same time, this is achieved by setting the number of *ackers* to 0 in the topology configuration as shown below:

```
1  Config config = new Config();
2  config.setNumAckers(0);
3  StormSubmitter.submitTopologyWithProgressBar(
4      "my-topology", config, builder.createTopology()
5  );
```

Line 1: Creates a new Storm configuration object.
Line 2: Sets the number ackers to 0, turning off Storm's SR fault-tolerance.

Line 3: Submits a topology called *my-topology* with the created configuration.

In order to make the workings of the system visible for testing, problem diagnostics and experiment purposes entries are written to the Storm log files when components are created, perform a checkpoint or are restored. Logging is performed using the logging framework supplied by Storm, which is based on the Apache Log4j logging library (*Log4j 2 Guide - Apache Log4j 2*, no date).

8.4.2. Class descriptions

The following class list describes some of the main classes that I designed for R+SM in Storm.

Class name	Description
`BufferingBoltDecorator`	Decorates a bolt object to add upstream backup and tuple replay functionality. Conforms to the standard bolt interface supplied by Storm.
`BufferingOutputCollector`	Decorates a Storm `OutputCollector` object. This object is provided by `BufferingBoltDecorator` to an application bolt whose output should be buffered. Emitted tuples are buffered before forwarded to their target component.
`IBufferState`	Interface for buffer state data structure. Provides operation to get buffer state for a specified downstream task id
`IHasBufferState`	Interface implemented by a bolt to expose buffer state in a standard format.
`IHasProcessingState`	Interface implemented by a bolt to expose processing state in a standard format. Provides operations for retrieving and restoring processing state
`ProcessingState`	Encapsulates processing state and associated timestamp of a stateful bolt.
`RecoverableBoltDecorator`	Decorates a bolt object to add checkpoint and recovery functionality. Conforms to the standard bolt interface.
`StringSerialzer`	Serializes an object to a string, and de-serializes a string into an object
`TopologyUtilities`	Provides constants and utility operations for working with tuples and streams.
`TupleBuffer`	Encapsulates upstream tuple buffer. Provides operations to add tuples to the buffer for a specified downstream task, trim tuples from the buffer for a task according to a specified timestamp and to retrieve tuples buffered for a specified downstream task.

`WindowActionTracker`	Tracks time within a windowed topology to indicate when tasks should be performed. Inspired by Storm's `SlidingWindowCounter` and `SlotBasedCounter` classes.
`WithTimestampDeclarer`	Provided to a decorated application bolt by `BufferingBoltDecorator`. Adds the declaration of a timestamp field to the default output stream of the decorated bolt.

8.4.3.Implementation

This section explores the implementation of following key areas of the system and discusses alternative approaches that I considered.

- Tuple capture
- Decorators vs base classes
- Buffer state implementation
- Processing state implementation
- Performing tasks at intervals

8.4.3.1. Tuple capture

When deciding where and how to buffer tuples for upstream backup the main considerations were the availability of information such as tuple routing and ease of implementation. The main options were within the implementation of the `emit` method of Storm's `OutputCollector` class, which is written in Clojure and defined within the source code of the executor daemon, or within the bolt whose output should be collected – this is the Java code that is a client of `OutputCollector`.

The advantage of implementing within the executor is that it provides access to all properties of a tuple and its associated state as well as the output queue and routing. The Java layer has a simpler but more restrictive API. When a tuple is emitted to the output collector it is emitted as a simple collection of values. The implementation of `OutputCollector.emit(…)` instantiations the tuple giving it an id, it also retrieves the routing for the tuple.

The advantage of implementing the tuple capture at the Java API layer of Storm is that the code there is thoroughly documented, easier to deploy (see section 8.4.1) and less likely to cause broad issues across the entire framework as a result of dependant code not being updated.

As I did not know whether all of the required tuple information would be available within the Java layer, initially this work was carried out within the executor with intention of storing the tuples on the task objects so that they could be accessed by the bolt for buffer trimming and replay tuples.

The existing Storm source code instantiated the tuple object within the parameter list of the function that places it in the executor's outbound buffer. This is shown in the extract of Storm's code below:

```clojure
(transfer-fn t
    (TupleImpl. worker-context
        values
        task-id
        stream
        (MessageId/makeId anchors-to-ids)))))
```

This was modified to instantiate the tuple within a let block, binding to a variable so that it could be sent to the upstream backup before being passed to the transfer function.

```clojure
(let [ts-out-tuple (TupleImpl. worker-context
                       values
                       task-id
                       stream
                       (MessageId/makeId anchors-to-ids))]
    (ts-buffer-output t ts-out-tuple bolt-obj)
    (transfer-fn t ts-out-tuple)))
```

The `ts-buffer-output` function has access to the task that emitted the tuple (`t` in the above code listing) and the instantiated tuple itself so it could either hold a buffer locally or store it in a buffer on the task that originally emitted as shown below – note that this example would require an `addToBuffer` method to be added to the bolt class:

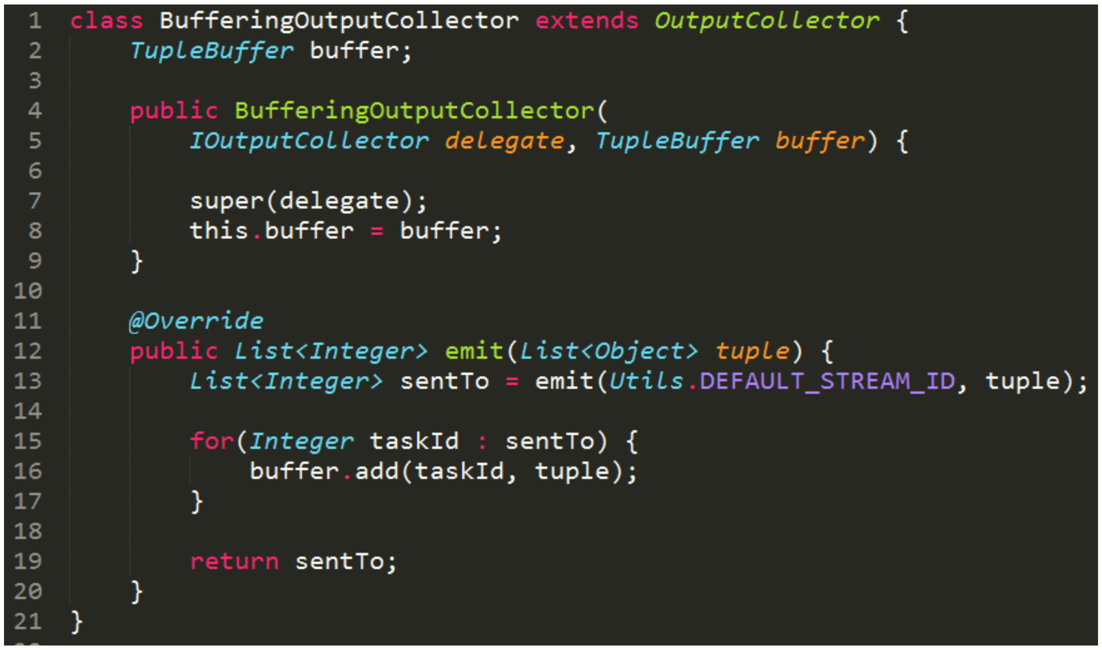
```clojure
(defn ts-buffer-output [target-task-id tuple bolt-obj]
    (.addToBuffer bolt-obj target-task-id tuple)))
```

As understanding of Storm grew, the decision was taken to implement at the Java layer as all of the required information was found to be available and communication between the bolts was easier. Communication between bolts is easier from this layer because tuples can be used as communication and Storm will handle routing. This decision was vindicated by the fact that other higher level abstractions have been built on top of Storm by implementing at the Java layer most notably Storm Trident (*Trident Tutorial*, no date).

When implementing tuple buffering at the point that it is emitted by the application bolt it was not desirable to directly modify the source code of that bolt as it would be impractical to do the same implementation for every application bolt and make maintenance more difficult when working on the experiments.

The solution was to implement a subclass of the `OutputCollector` class that the application bolt emits tuples through. The emit method first passes the emitted tuple to the real output collector and then stores it in a buffer.

A simplified example of the above is shown in the following listing:



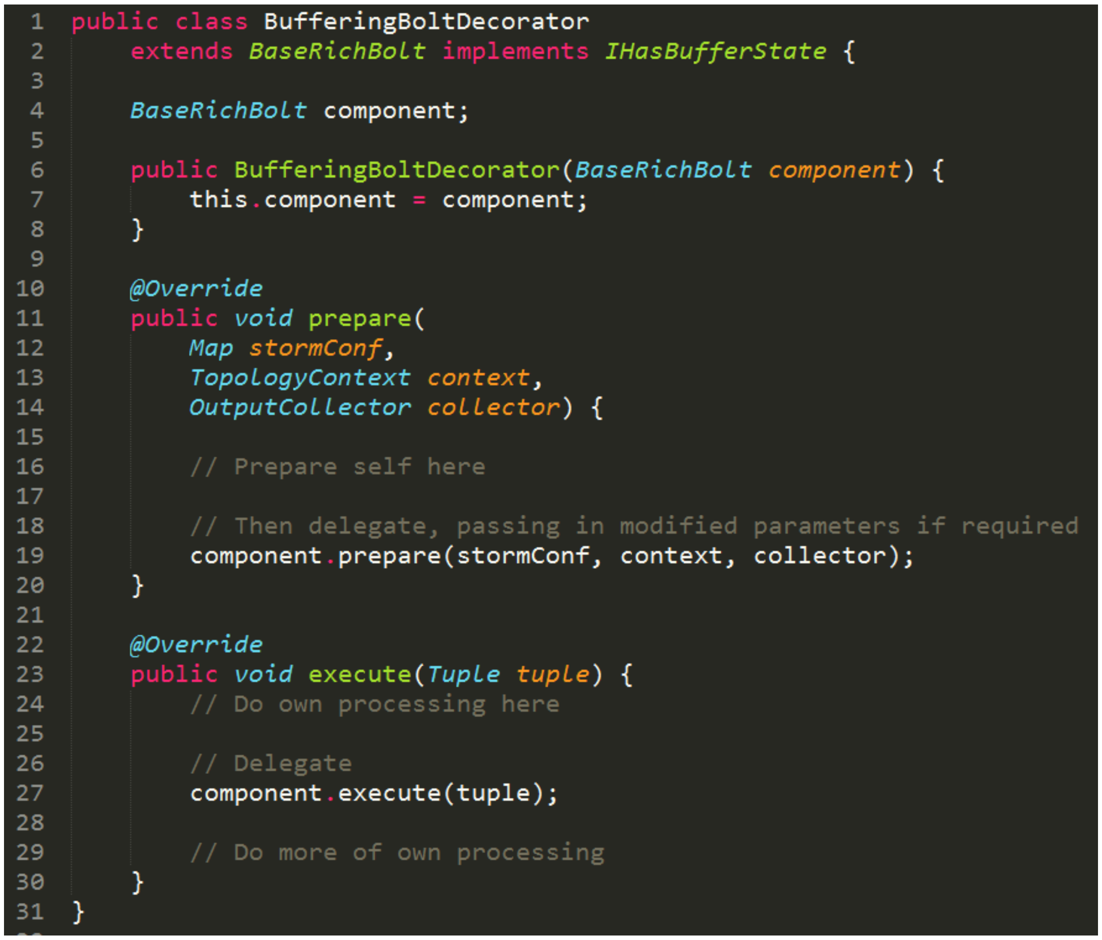
```java
class BufferingOutputCollector extends OutputCollector {
    TupleBuffer buffer;

    public BufferingOutputCollector(
        IOutputCollector delegate, TupleBuffer buffer) {

        super(delegate);
        this.buffer = buffer;
    }

    @Override
    public List<Integer> emit(List<Object> tuple) {
        List<Integer> sentTo = emit(Utils.DEFAULT_STREAM_ID, tuple);

        for(Integer taskId : sentTo) {
            buffer.add(taskId, tuple);
        }

        return sentTo;
    }
}
```

In this approach the buffered tuple is still a List object, not an instantiated Tuple. This means that the tuple is not available to the buffer as it has not yet been created; however presently the tuple id is not required. As discussed below Storm's Tuple class does not have a timestamp member, however if this were to be added it would also not be available to the `BufferingOutputCollector` class.

8.4.3.2. Decorators and base classes

This section discusses the reason that I selected the decorator pattern for the design and describes an alternative design that was also trialled.

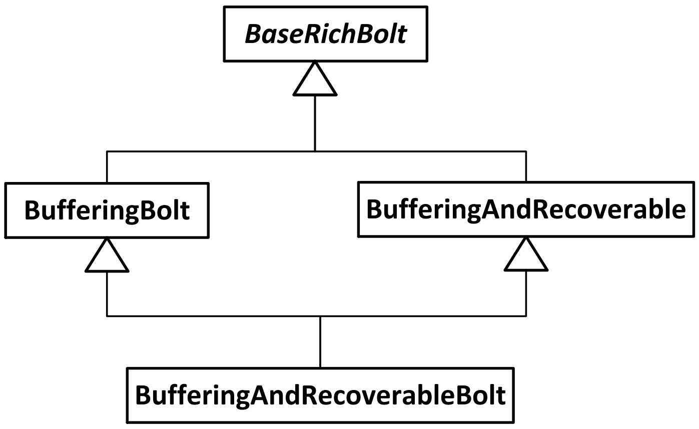
The decorator pattern is described in section 3.2 and its application in the design of `BufferingBoltDecorator` and `RecoverableBoltDecorator` is described in section 5.4.1. The following significantly simplified example shows how the `BufferingBoltDecorator` both confirms to Storm's `IBolt` interface and also delegates calls to the application bolt. In this code snippets the `BaseRichBolt`, `TopologyContext`, `OutputCollector` and `Tuple` classes are supplied with Storm. The `Map` class is supplied with Java.



```java
public class BufferingBoltDecorator
    extends BaseRichBolt implements IHasBufferState {

    BaseRichBolt component;

    public BufferingBoltDecorator(BaseRichBolt component) {
        this.component = component;
    }

    @Override
    public void prepare(
        Map stormConf,
        TopologyContext context,
        OutputCollector collector) {

        // Prepare self here

        // Then delegate, passing in modified parameters if required
        component.prepare(stormConf, context, collector);
    }

    @Override
    public void execute(Tuple tuple) {
        // Do own processing here

        // Delegate
        component.execute(tuple);

        // Do more of own processing
    }
}
```

The above allows the decorator to modify the parameters that are used to prepare the decorated component, this is utilised to pass in a modified `OutputCollector` that buffers emitted tuples.

An alternative to this approach that was also considered was the use of base classes instead of decorators. These base classes would themselves be sub-classes of `BaseRichBolt`. An application bolt may need to act as an upstream backup, be recoverable or both. However Java supports only single inheritance, as such multiple base classes would be required to support the three cases. For example the base classes might be structured as shown in the following UML class diagram:



The decorator approach achieves a similar result with less classes and is more flexible since application bolts do not need to be hardcoded as generalisations of specific base classes. A topology developer can change application bolt roles by composing different classes together.

Whether using base classes or decorators it was important that the fault-tolerance logic was separated from application logic. This made the development of the experiments easier especially since a large number of changes were made to the experiments as greater insight was gained into how they should run. These changes could be made without impacting the fault-tolerance implementation.

8.4.3.3. Tuple buffer implementation

The tuple buffer is encapsulated as class `TupleBuffer`, which provides useful operations for working with it. It is implemented as the data structure:

```
IntMap<TreeMap<DateTime, List<Values>>>
```

The key to `IntMap` is the id of the downstream task that the tuple was sent to. This allows efficient retrieval, for trimming and replay purposes, of all the tuples that were sent to a specific task. The `IntMap` is pre-allocated on creation with the correct downstream task ids.

`IntMap` is a utility class from the ***Kyro*** (*EsotericSoftware/kryo*, no date) serialization library used by Storm. An alternative to `IntMap` was to use Java's `HashMap` with an Integer as it's key (e.g. `HashMap<Integer, … >`). `HashMap` does not support using `int` as a key because it is a primitive, however the task id is of type int. Using Integer as the map key would cause boxing and unboxing of the key when creating new entries and retrieving existing entries, which carries a performance penalty.

Tuples for each task are stored according to their created timestamp which is the key in an object of type `TreeMap` which is supplied with Java. `TreeMap` orders entries by the natural order of keys which has several benefits in this application. Firstly it facilitates replay of tuples in their original order as the map can be iterated. Secondly tuples with timestamps before a certain timestamp can efficiently be trimmed using the `tailMap(K fromKey)` method which returns all entries greater than the supplied parameter

8.4.3.1. Processing state implementation

Processing state is encapsulated as class `ProcessingState`, it includes a timestamp member, which is the timestamp of the most recent tuple represented in the state, and a member of type Object for the state. The use of the Object type makes the class able to support any state object that an application bolt may hold provides that the object implements Java's `ISerializable` interface. Implementation of this interface enables serializing to a string for storage in Redis.

8.4.3.2. Performing tasks at intervals

Within fault-tolerance decorated classes (see section 5.4.1) and the experiment classes there is a requirement to perform actions at certain intervals. For example the R+SM implementation needs to perform a checkpoint every *n* seconds and the experiments need to fail at a certain point within a window.

`WindowActionTracker` is designed to be configurable to support these various requirements. It's design is influenced by Storm's `SlidingWindowCounter` and `SlotBasedCounter` classes. When instantiated it internally assigns a slot per tick that will occur within a window – for example if tick tuples occur once per second and there is a 10 second window then `WindowActionTracker` will assign 10 slots. Its update method is called every time a tick tuple is received so that it can update its tracker of the current slot. When it reaches the end of the window, the tracker loops round to refer to the first slot of the window again, this is achieved using modulo as shown in the extract below. The implementation of this part of the code was influenced by the `slotAfter` method of Storm's `SlidingWindowCounter` class.

```java
public void update(Logger log) {
    current = (current % slots) + 1;
    if (current == 1) {
        log.info(name + " window start is now at slot 1");
    }
}
```

The class offers the ability to be notified of something that needs to be carried out at specific slot in the window via the `shouldPerformOncePerWindowAction` method, which does a simple check to see if the current window is the same as the configured slot that the action should be carried out in.

```java
public boolean shouldPerformOncePerWindowAction() {
    return current == actionAt;
}
```

Notification of when to perform actions that occur repeatedly within the window, such as to perform a checkpoint, is provided via the `shouldPerformRepeatableAction` method.

```java
public boolean shouldPerformRepeatableAction() {
    return current != 0 && current % actionAt == 0 ;
}
```

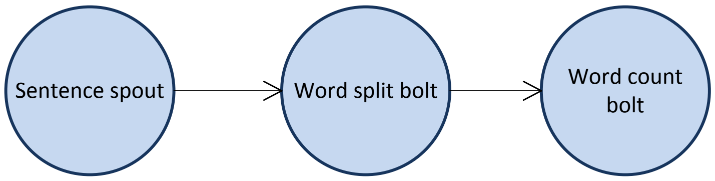
The distinction here is that if the `WindowActionTracker` is configured with a 10 second window and 10 slots with `actionAt = 2`, `shouldPerformOncePerWindowAction` will return true, only at slot 2, whereas `shouldPerformRepeatableAction` will return true at slots 2, 4, 6, 8 and 10.

9. Results

This chapter begins with an overview of experiments performed. Experiment results along with associated discussion and conclusions are split into to two phases. The final section of this chapter explores some of the issues encountered when running the experiments.

9.1. Overview of experiments

In order to evaluate performance of SR and R+SM in Storm, I created an experimental word-count topology to test each fault-tolerance approach. The topologies consist of three components - a spout and two bolts, these components have different implementations for the SR and R+SM topologies.



Each time a new tuple is requested by Storm, the ***sentence spout*** selects one from a list of five pre-defined sentences and emits it as a new tuple. The SR version of the sentence spout also keeps a list of all emitted tuples. When its `ack` method is called it, and passed a tuple id, it removes that tuple from the list. If instead the spout's `fail` method is called, the spout will replay that tuple back into the topology on stream ***ts_recovery_stream*** when a new tuple is requested. The tuple is replayed on the recovery stream so that it can be identified as a replayed tuple in the logs when establishing how long recovery took.

The ***word split*** bolt splits a sentence into separate words using the space character as the delimiter and emits the words as separate tuples. The implementation is similar between the SR and R+SM versions, however the SR must acknowledge each input tuple. The acknowledgment is performed automatically by extending the `BaseBasicBolt` class, rather than `BaseRichBolt` class that is used by the R+SM version.

The ***word count*** bolt receives words as input and maintains a count of the number of time each distinct word is received within a 30 second tumbling window. At the end of each window, the word count bolt emits the counts for each word, and then clears its list of the counts ready for the start of the next window. A window is required because the SR can only recover stateful operators whose state relies on a limited time period of tuples. The query must manually acknowledge each tuple within a tuple timeout period (30 seconds by default in Storm) and any tuples that are not acknowledged within the timeout will automatically be marked as failed and replayed. However once acknowledged, tuples will be removed from the source and cannot then be replayed in order to recover a stateful operator. Further discussion of this constraint is provided in section 8.3. R+SM does not suffer from this limitation.

The SR version of the word count bolt must keep a list of tuple ids that are currently reflected in its processing state so that it can acknowledge them when the portion of the processing state that reflects them is aged-out. The R+SM version of the word count bolt does not need to track and acknowledge tuple ids, however it does have to implement the `IHasProcessingState` interface, which declares operations for getting and setting the processing state. This is interface is used by the `RecoveringBoltDecorator` object that decorates the word count bolt in the R+SM approach.

I performed the experiments with a single task per component and with debug enabled in the topology configuration. Enabling debug increased the amount of diagnostic information available in the log files, which is useful for the metrics described later in this section. The Storm cluster ran in distributed mode. Each node was run within a separate ***Docker*** (*Docker - Build, Ship, and Run Any App, Anywhere*, no date) container and all containers hosted on a single physical server. I used ***Storm Docker*** (*wurstmeister/storm-docker*, no date) and ***Fig*** (*docker/fig*, no date) to orchestrate the Docker containers, allowing the entire cluster to be started using the command `fig up -d`. The server has a 4 core Intel Xeon E5-2407 2.20GHz processor, 8GB of RAM and runs Ubuntu Linux 12.04LTS 64bit.

I recorded the following metrics to the measure efficiency of each fault-tolerance approach:

- ***Recovery time***
- ***Processing latency***
- ***Throughput***

Recovery time is a measure of how quickly the fault-tolerance approach is able to recover a failed task. It is the time from when a failed task is ready to be recovered to the time that the final replayed tuple arrives at the recovering task. For the purposes of these experiments I do not consider the time that is takes for Storm to identify a failed task and re-created a replacement to be part of the recovery time, instead the recovery time begins when Storm has created a failed task and it is ready to be recovered by either SR or R+SM. The word count bolt writes a record to the log before failing, this happens whether failing by throwing an exception as described in phase 1 below, or by performing a pseudo fail and throwing away state, as described in phase 2. When running with debug on, Storm automatically writes a log message for every tuple received by a task. The logs can be searched with command line tools such as ***grep*** (*GNU grep - GNU Grep: Print lines matching a pattern - GNU Project - Free Software Foundation*, no date) and ***tail*** (*tail invocation - GNU Coreutils*, no date) to find the time of arrival of the last tuple on the recovery stream.

Processing latency is the time that it takes for a tuple tree to be processed by the topology, it shows us the runtime overhead of the fault-tolerance approach in use. In the word count topology tuples only travel from the bolt that emitted them to the next bolt in the topology, rather than from the spout all the way to the word count bolt. For example sentence tuples emitted from the spout travel to the word split bolt, the word split bolt then emits a new tuple for each word in the sentence. In order to measure processing latency over the entire topology I generate a timestamp when a tuple is created at the spout, when new tuples are created later in the topology, for example word tuples emitted by the split bolt, the timestamp is passed from the source tuple being created. In the word count bolt the difference in milliseconds between the current timestamp and the created timestamp of each tuple is calculated and written to the log files.

Throughput is the number of tuples flowing into the topology; it is useful because it shows when the number of tuples that can be processed is reduced – for example during recovery of an operator. Storm automatically records the number of tuples that are emitted by each component every 60 seconds, including spouts. When running the experiment topologies, I register Storm's built in ***LoggingMetricsConsumer*** class; this causes the automatically recorded metrics to be written to a log file called metrics.log.

Throughout is dependent on the tuple input rate of the topology. Since Storm requests new tuples (pulls) rather than being pushed them, it was not possible to exactly control input rate and specify a specific number of tuples to input per second. Instead input rate was controlled by implementing a pause in each call to the `nextTuple` method of the spout. A pause of 10 milliseconds yields an input rate of approximately 95 tuples per seconds, and smaller pauses yield higher rates.

9.2. Experiment phase 1

This section presents an initial phase 1 of experiments that generated useful test data with regards to processing overhead of R+SM and SR. The following section focuses more closely on recovery time and includes a larger set of results.

The aims of this phase of experiments were:

- Determine if SR or R+SM is faster at recovering a task in a three component topology as described in previous section.
- Determine which approach has the least impact on processing latency and allows for maximum throughput

I configured the topologies with a 30 second tumbling window. The R+SM topology was configured to checkpoint every 30 seconds. The sentence spout was configured to pause for 10 milliseconds each time a tuple was emitted, resulting in a tuple input rate of approximately 95 tuples per second (see section 9.1 above for explanation).

To create a failure, the word count bolt was configured to throw an exception 15 seconds into a processing window.

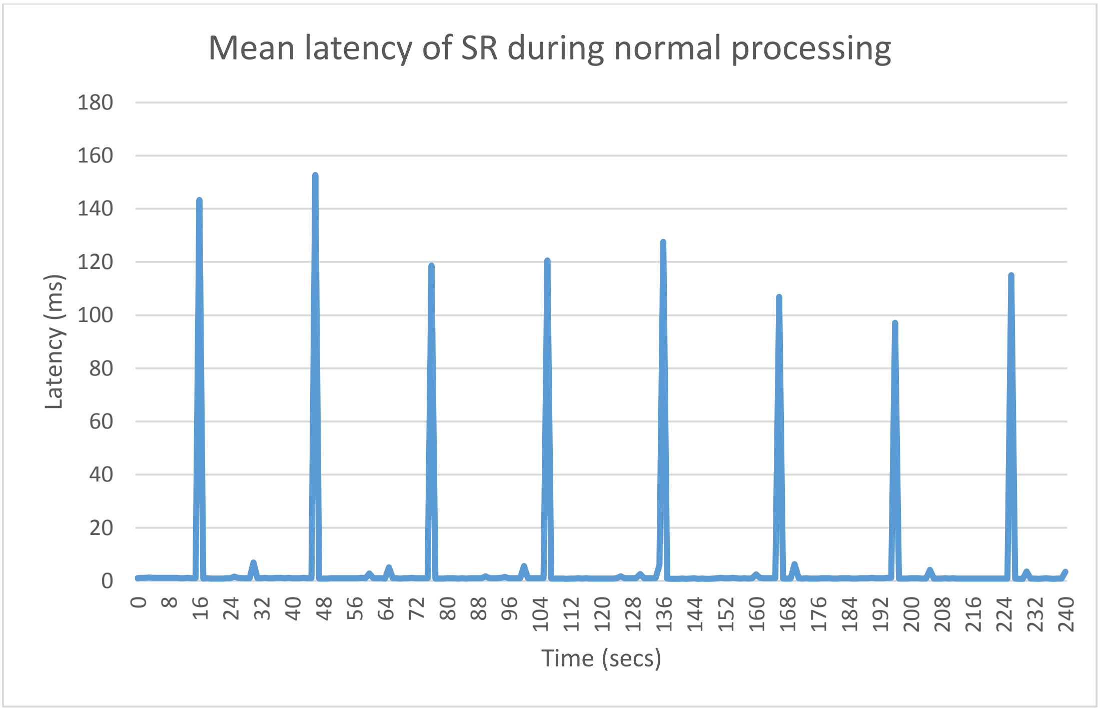
The following table shows the recovery times for SR and R+SM. R+SM took 31 seconds longer to recover the failed task.

Approach	Recovery time
SR	00:01:03
R+SM	00:01:34

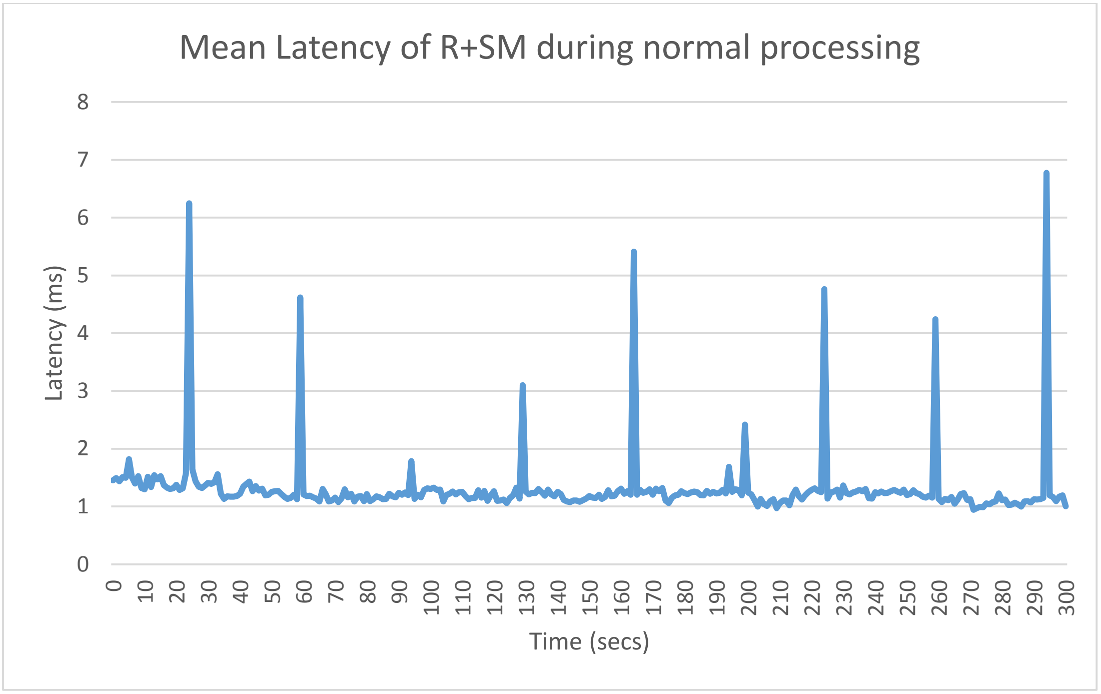
The following table shows a summary of tuple latency during a 4 minute period of normal running of the topologies, before any failure or recovery took place. The median and 95th percentiles are the same between the two approaches, but we can see a higher mean and a significantly higher standard deviation for the SR approach.

Approach	Median (ms)	Mean (ms)	STD (ms)	95th percentile (ms)
SR	1	6.36	46.27	2
R+SM	1	1.32	2.33	2

The higher mean and standard deviation for the SR approach can be investigated further by plotting a line chart of the mean latency. The following chart shows mean tuple latency per second of the SR approach before any failure/recovery.



The following chart shows mean tuple latency per second of the R+SM approach before any failure/recovery.



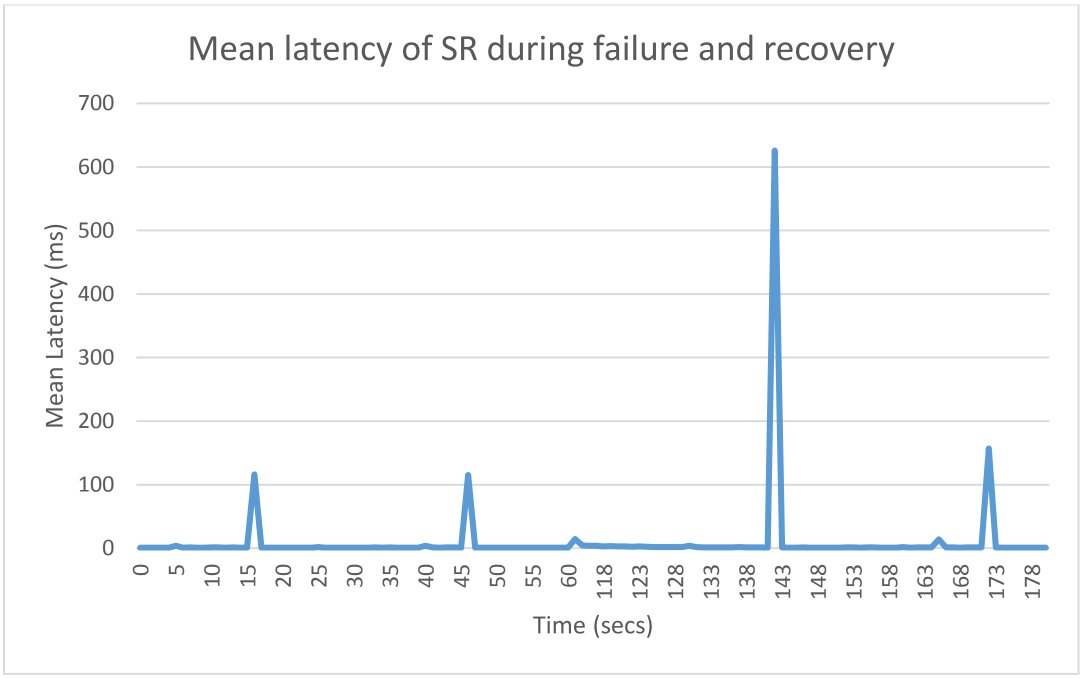
We can see that both approaches have spikes in tuple latency approximately every 30 seconds, the 30 second intervals is likely to correlate with the processing window. The spikes are significantly larger for the SR approach, it is possible that these spikes in latency occur because the word count has to acknowledge 30 seconds worth of tuples when they are no longer reflected in the processing state at the end of the processing window. Since under R+SM the word count bolt only needs to perform a checkpoint (accounting for the slight spikes in latency), there is no such effect on tuples flowing through the topology.

The following table shows a summary of tuple latency for the SR approach from the time of failure to the time that the failed task is fully recovered.

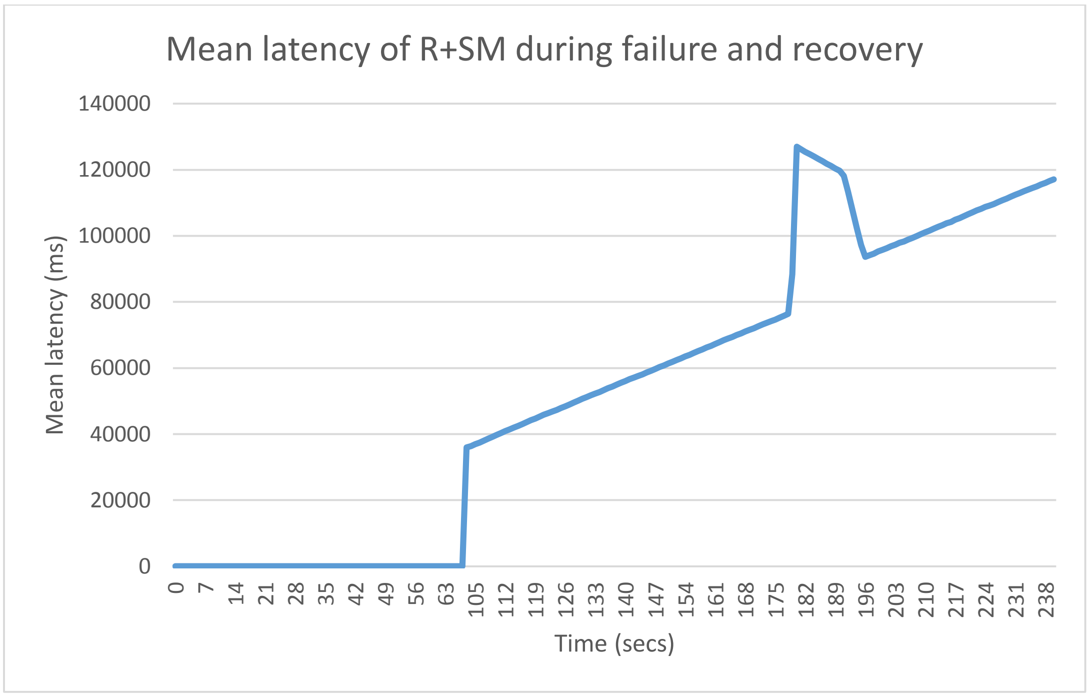
Approach	Median (ms)	Mean (ms)	STD (ms)	95th percentile (ms)
SR	1	11.85	86.76	3
R+SM	163895	162392.98	77207.76	278992

Here we can see that all measures are very significantly higher for R+SM indicating an underlying issue with the implementation of R+SM.

Plotting graphs of the mean tuple latency per second from 1 minute before failure, through recovery to 1 minute after failure gives us more insight into the how the data are behaving.



The chart for SR shows a large spike in tuple latency around the time of the failure and recovery, but latency returns to normal after the failed task has recovered



The R+SM chart however shows that tuple latency increases steadily following failure of the task. A spike is also visible, this can be accounted for the fact that in order to recover the failed word count task, the upstream backup will replay all tuples that are not present in a checkpoint (30 seconds worth since in this experiment checkpoints are not being performed within a processing window).

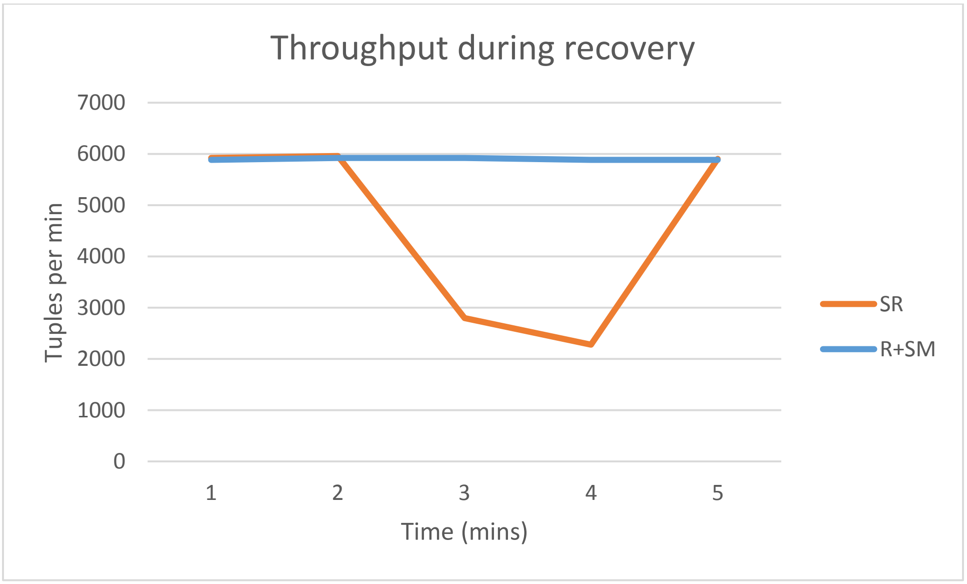
In the test topologies, the sentence spout does not stop emitting tuples when a task fails or is being recovered. However under SR the tuples are replayed from the spout; whilst tuples are being replayed no new tuples will be entering the topology (only replayed tuples). By contrast under the R+SM implementation new tuples are always entering the topology. During the period when the word count task has failed and is being re-created by Storm, tuples will still be entering the topology and filling the outbound buffer and transfer queue of the word split operator.

We can see the number of tuples emitted by the sentence spout by looking at the throughput metrics for the period from 1 minute before failure, through recovery to 1 minute after recovery. Note however that the throughput metrics are recorded by Storm only once every 60 seconds, so figures in the table and charts for throughput are less granular than for latency data.

Approach	Median (tuples per min)	Mean (tuples per min)	STD (tuples per min)	95th percentile (tuples per min)
SR	4360	4240	1709.97	5954
R+SM	5880	5896	19.60	5920

The summary shows that more tuples released into the R+SM topology over the course of the failure and recovery.

The following line chart shows the mean number tuples released into the SR and R+SM topologies per minute over a 5 minute period in which the word count task fails and was recovered. The chart shows that the under SR the rate of input of new tuples was reduced during recovery. This is due the spout emitting replay tuples rather than new tuples.



This phase of experiments shows that the latency is lower under the R+SM approach during normal processing. This can be accounted for by the fact that under SR, the word count bolt must maintain a list of all tuples received during the last window and acknowledge each of them individually at the end of the window. Since every tuple acknowledgment results in an additional tuple being sent to the spout, there is a noticeable overhead in the results.

This phase of experiments also highlighted a critical issue with the R+SM implementation, where tuple latency increased dramatically following failure. Further the tuple latency continued to increase until measurements were ceased. This issue is likely to be caused by two factors. Firstly, when the word count task failed there was a delay before it was recreated and ready to have its state recovered. During this period, new tuples were flowing into the topology but could continue to be processed past the word split bolt, possibly resulting in a build of tuples that caused Storm's internal buffers to be overloaded. Secondly, at the time of failure, 30 seconds worth of tuples (approximately 2850) were stored at the upstream backup. In order to recover the word count task's state, these buffered tuples were replayed. However releasing such a large number of tuples when Storm's buffers were likely to already be under pressure is likely to have compounded the issue.

The results also show that the SR approach was able to perform recovery faster than R+SM under these conditions. However the recovery time for R+SM is likely to have been significantly affected by the high tuple processing latency.

9.3. Experiment phase 2

I made several changes to the experiments for this phase, with the intention of focusing more closely on the time taken to recover the state of the failed task. Whilst re-creation of the failed task was not in within the scope of the previous experiments, it still had a bearing on the results. The delay between the task failing and being recreated caused tuples to build up in the outbound queues of the word split bolt, this meant that tuples replayed in order to recover the word count bolt could not reach it as quickly and thus recovery took longer. Section 9.4 describes further issues related to Storm's recreation of the failed task.

The aim of this experiment was to determine if SR or R+SM is faster at recovering a failed task in a three component topology.

Experiments were performed with a 30 second processing window and fail times of 5, 15 and 30 seconds within the window. The R+SM approach was configured to checkpoint every 5 seconds. Tuple input rates of approximately 10 and 20 per second were configured using pauses in the sentence of spout of 100 and 50 milliseconds respectively. These input rates were significantly lower than phase 1 in order to reduce the possibility of having issues with increasing latency as seen in that phase and focus on the time taken to successfully perform a recovery.

I also modified the failure mechanism for this phase of experiments. Rather than throwing an exception to force the entire executor thread to die, the word count task performed a pseudo failure by throwing away its state. This made the task's state be the same as it would be if it had just been re-created. For the SR word count bolt, the changes involved clearing all word counts, and clearing the list of tuples pending acknowledgement; recovery under SR then commenced automatically as the unacknowledged tuples timed out. The R+SM word count bolt changes involved clearing all word counts, and then triggering recovery by retrieving processing state, and requesting recovery by the upstream backup.

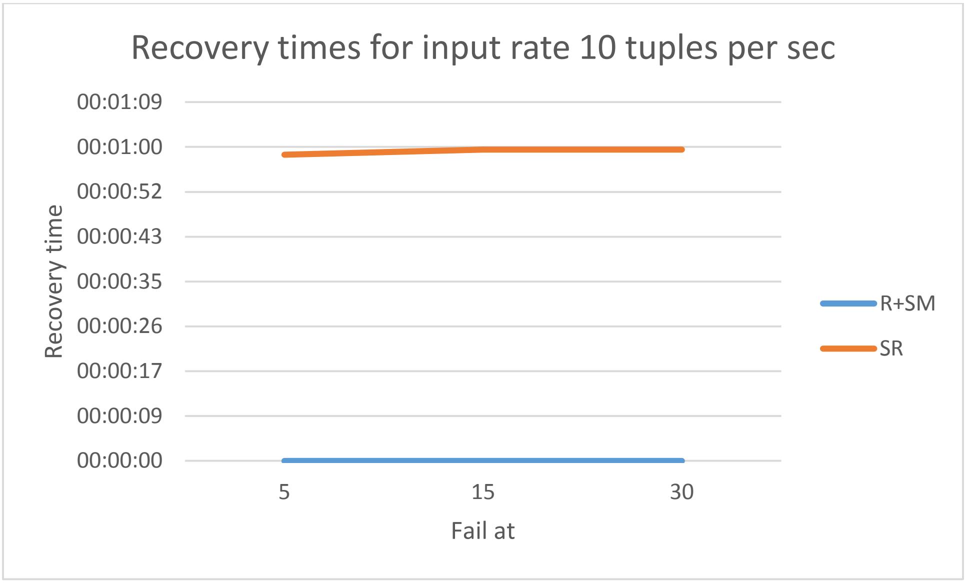
The changes to the fail mechanism meant that I was able to measure the exact time taken to perform recovery of a task that was in a state ready to be recovery, avoiding interference caused by a true failure and re-creation of the task. The definition of recovery time remained unchanged in that I continued to measure from the time that task was ready to be recovered, to the time that it was full recovered.

Table N shows an overview of the recovery times of SR and R+SM with variable input rates, and failure times within a window.

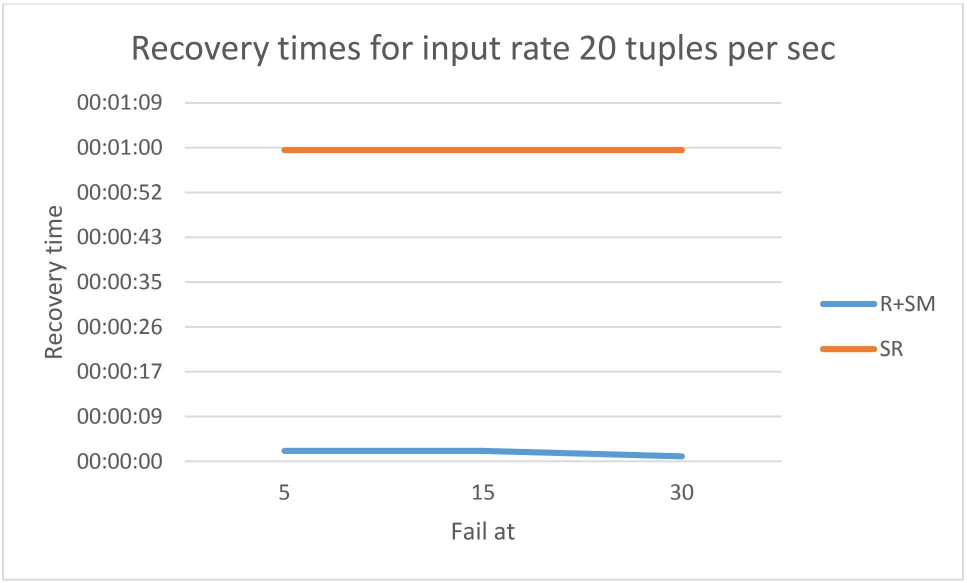
Approach	Tuples per sec (approx)	Fail at time (secs)	Recovery time (secs)
R+SM	10	5	00:00:00
SR	10	5	00:00:59
R+SM	10	15	00:00:00
SR	10	15	00:01:00
R+SM	10	30	00:00:00
SR	10	30	00:01:00
R+SM	20	5	00:00:02
SR	20	5	00:01:00
R+SM	20	15	00:00:02
SR	20	15	00:01:00
R+SM	20	30	00:00:01
SR	20	30	00:01:00

Note that some recovery times for R+SM show as 0 seconds. These recovery times were actually greater than 0 seconds but less than 1 second. The log files used to deduce recovery times contain timestamps only to second precision rather than millisecond precision that would allow us to see the exact time taken for these recoveries.

The following chart shows recovery times for SR and R+SM with a tuple input rate of approximately 10 tuples per second.



The following chart shows recovery times for SR and R+SM with a tuple input rate of approximately 20 tuples per second.



The results show that R+SM performed recovery consistently faster than SR. SR took around 1 minute to recover the failed task, for all fail times and input rates. For R+SM, recovery times are marginally higher for the experiments with input rates of approximately 20 milliseconds than those with input rates of approximately 10 milliseconds.

The following table shows tuple processing latency during a 4 minute period of normal processing (no failure or recovery) for both approaches.

Approach	Tuples per sec (approx)	Median (ms)	Mean (ms)	STD (ms)	95th percentile (ms)
SR	10	2	3.39	14.13	4
R+SM	10	2	2.17	0.74	3.3
SR	20	1	1.78	8.06	2
R+SM	20	2	1.62	0.73	2

Looking at the above summary of tuple processing latency we can see that the mean latency for SR is slightly higher than R+SM and the STD is significantly higher. As seen in phase 1, this can be accounted for by the requirement to acknowledge an entire window (30 seconds) of tuples at the end of each window under SR. Interestingly tuple latency is shown to decrease with higher the higher input rate.

The following table summarises tuple processing latency during the period of recovery for each experiment.

Approach	Tuples/s (approx)	Fail time (secs)	Median (ms)	Mean (ms)	STD (ms)	95th percentile (ms)
R+SM	10	5	2133	2173.11	1613.00	4725
SR	10	5	2	1.87	3.67	2
R+SM	10	15	2122	2194.40	1520.47	4654.6
SR	10	15	2	2.27	3.31	3
R+SM	10	30	2349	2346.02	1551.10	4742
SR	10	30	2	2.60	6.97	3
R+SM	20	5	2038	2058.65	1477.42	4579
SR	20	5	1	1.83	9.38	2
R+SM	20	15	3020	2879.38	1444.93	4910
SR	20	15	2	2.00	7.87	2
R+SM	20	30	2438	2491.25	1460.33	4778
SR	20	30	2	2.05	8.12	2

These data show that whilst tuple processing latency was lower overall for the R+SM, for the specific period of failure and recovery, processing latency was higher for R+SM than SR. However it should be noted that the time periods involved here are very different – for R+SM these data were recorded over a maximum of 2 seconds whereas the data for SR were recorded over a period of approximately 1 minute. The difference in time period is because these data only represent the period of recovery, and under R+SM this period of recovery was much smaller.

Overall in this phase of experiments, R+SM was consistently faster that SR at recovery of the failed word count task. This is likely to be because under SR each tuple has to timeout independently. Under both SR and R+SM the time to recovery was not significantly affected by the tuple input rate, though this was reasonably low for all experiments.

Under SR the SPS has to perform more work to recover a failed operator the later in the processing window it fails. This is because all tuples that were sent during the processing window must be replayed, so the longer ago the start of the processing window, the more tuples have to be replayed. As such, I expected the time to recovery under SR to be larger for later fail at times. This is not shown in the results – recovery time is relatively constant as the fail at time becomes later; this may be because of the low tuple input rate. Further work would be useful to perform more experiments with gradually higher tuple input rates to see the effect on recovery times.

The results show that processing latency was higher during the recovery period for both SR and R+SM and this is understandable given the additional work that is performed to recover a failed task using either approach. Also of interest in this phase of experiments was the fact the tuple latency during normal processing was shown to be marginally lower when tuple rates increased. This may be due to the performance of internal buffers.

9.4. Issue encountered

A number of challenges were encountered during the running of the experiments. Firstly the R+SM approach was found to suffer from continuous increases in tuple latency following failure.

The continuous increases in tuple latency when recovering with the R+SM approach are discussed in section 9.2 above and a number of possible causes are identified – namely the build-up of tuples in the outbound buffer of the word split task and the large number of tuples that are replayed as part of R+SM in order to recovery the word count task. In order to avoid this issue tuple input rates were reduced for the second set of experiments and the approach to simulating failure also made.

Other possible techniques that were investigated and may have been useful were to increase the size of the outbound buffers of the word split bolt, and also to replay tuples from the upstream backup more slowly.

The outbound buffer of the word split bolt is a Disruptor queue (see section 8.2) with a default size of 1024; this is controlled by the configuration setting `topology.executor.send.buffer.size`. I tried increasing this to 8024 and found that the latency issue persisted, however I still feel that this configuration change could be useful if continuing to develop the R+SM approach.

Replaying tuples more slowly could help with the tuple latency as it would prevent the topology being further overloaded by the replay of tuples for the recovery. The issue with this approach is that under some circumstances tuples could be replayed as fast as possible (as shown in phase 2) and under different circumstances, when latency is increasing, the tuples should be replayed more slowly. It should be noted that since Storm uses a separate thread to read from the executor send buffer, emitting all replayed tuples in a single pass, should not block tuples from being transferred.

The second issue I encountered whilst running the experiments was with the way Storm identifies and re-created failed tasks. By looking at the logs I noticed that word count task was intermittently being re-created twice following a failure, rather than once which I expected. Close inspection of the log files revealed the following sequence of events occurring:

1. Word count task fail (as expected as part of the experiment)
2. Supervisor notices that the worker running the word count task is not responding and restarts it. We will hence forth call this worker 1.
3. Whilst the worker is being restarted
    a. Nimbus notices that the word count task is not responding. It instructs the Supervisor that worker 1 is invalid and re-allocates the word count task to the worker that is running the word split task. We will hence forth refer to this worker as worker 2.
    b. Having been told that worker 1 is invalid, Supervisor kills worker 1 (even though still in process of restarting from step 2)
    c. Having been told to re-allocate word count task, Supervisor starts the word count task on worker 2. Word split and word count tasks now both running on worker 2.
4. Supervisor starts worker 1 again
5. Nimbus sees that worker 1 is now valid and re-allocates the word count task back to it
6. Supervisor kills both the word count and word split bolts, word split is restarted on worker 2, word count is restarted on worker 1

The fact that the word count task was being re-created multiple times meant that recovery was taking longer than it needed to. However more critically, the restart of the word split task meant that the upstream backup was lost and recovery of the word count operator was no longer possible.

I spent approximately 4 weeks examining log files to identify the above sequence of events and investigating possible solutions.

Further investigation showed that the Supervisor and Nimbus daemons have timeout periods for controlling how long a task is allowed to not respond to a heartbeat before it is considered to have failed. The timeout periods are controlled by configuration settings `nimbus.task.timeout.secs` and `supervisor.worker.timeout.secs`; both of which default to 30 seconds.

In order to prevent Nimbus marking the worker has invalid whist it was already being re-created I changed `nimbus.task.timeout.secs` to 120 seconds. This meant that the Supervisor could identify the failed task and have time to restart the worker before Nimbus realised that the task had died.

The effect of the timeout change was as expected in that when the task failed, the Supervisor identified the failure after 30 seconds and then restarted the worker without interference from Nimbus. However further testing showed that when recovering with R+SM the replayed tuples were not reaching the recovering task; the logs showed them being emitted from the upstream backup but never reaching the recovering task. No further tuples were reaching the recovering task until approximately 1 minute later.

I thought that it was possible that recovering word count task was not ready to start receiving tuples straight away. In order to prevent the replayed tuples being lost due to being replayed before the recovering task was ready to receive them, I implemented a polling mechanism.

I modified `RecoverableBoltDecorator` so that if it received a tuple on stream `ts_poll_ready_stream` it would respond with a tuple on stream `ts_confirm_ready_stream`. I modified `BufferingBoltDecorator` to create a new thread to send tuples to the on stream `ts_poll_ready_stream` once per second. Upon receiving a response on stream `ts_confirm_ready_stream` the tuples from the upstream backup could be replayed.

When this was approach was tested I could see in the logs that the polling tuples were never received by the recovering word count task. Further, no tuples were ever received by the recovering task.

This issue related to how Storm re-created failed tasks and not directly to my main focus which was investigating the recovery of a task when it is ready to be recovered. As such I took the decision not to spend further time trying to solve this issue and instead focused on the experiments discussed in section 9.3.

10. Discussion

This chapter begins by covering each objective and discussing how the project met that objectives along with further work and recommendations.

Note that discussion of experiment results is included in the chapter 8 above.

Objective 1: Review existing research on fault-tolerance in SPSs.

Chapter 5 discusses the main SPS fault-tolerance concepts from the literature before going into more detail about R+SM and noting that whilst Castro Fernandez *et al.*, 2013 do evaluate the overhead of their approach, they do not compare the overhead of R+SM with alternative mechanisms. Section 6.1 provides further detail of R+SM.

Objective 2: Design and implement R+SM in Apache Storm.

This objective was successfully met as can be seen by the design presented with discussion in chapter 8 and the successful recovery of a 3 operator query as documented in chapter 9.

I was able to perform the required experiments using the components that I created and the implementation could be extended to a full implementation for release as an alternative fault-tolerance strategy in Storm.

My selection of implementation layer (as discussed in section 8.4.3.1) was the correct choice given the circumstances. Implementing at the Java layer reduced the complexity of the task (which was important because at any layer design and implementation was difficult), and also made best use of my existing skills.

One difference between my design and the Castro Fernandez *et al.*, 2013 was that they discuss the SPS performing all actions necessary to the approach, on behalf of operators. In Storm this would mean Nimbus and Supervisor performing checkpoints, restoring state etc. In my design the order of actions is the same but control is with the operators (since the decorator classes discussed in 8.4.3.2 are operators themselves); the `RecoverableBoltDecorator` initiates a checkpoint, rather than the Supervisor daemon initiating it.

The use of decorators to implement the main fault-tolerance was a success, it provided a flexible mechanism for introducing new functionality, in a way that integrated appropriately with Storm and maintained separation of concerns from the application logic. Whilst developing and making changes to the experiments, I rarely had to look at the decorator classes which allowed me to focus purely on the experiment logic.

However a number of changes would be required to continue to develop the implementation. The tuple timestamp that I introduced to tuple payloads would need to become a true property of all tuples, this would require changes to Storm's core. Making this change would make reduce the amount of code – and therefor the complexity, and the also be likely to improve performance.

The main area of functionality that would need be extended if extending my implementation for a full release in Storm is the replay of tuples. Rather than replaying tuples from an upstream backup more slowly so as not to overload the SPS. This is discussed in section 9.4.

The current design also has several limitations. Firstly it does not provide a mechanism to recover a failed spout however limitation is shared by the description of R+SM presented in (Castro Fernandez *et al.*, 2013, p. 3) – they state that *"we assume that sources and sinks cannot fail".* Note that is Storm, spouts often read from a queue service, when a tuple is acknowledged as being processed the spout can remove that source data from queue; in these circumstances Storm's SR approach can recover from a failed spout.

A second limitation is that the first bolt in a topology is not recoverable. This is because Storm spouts do not have the necessary operations to be able to receive arbitrary tuples. They can receive specific types of tuples to indicate acknowledgement and failure of tuples but this is available only to Storm SPS. Allowing recovery of the first bolt in a topology could be achieved by extending the Storm's spout.

Objective 3: Design and perform experiments to collect performance data for R+SM vs SR approaches to recovery.

The design and implementation of the experiments was successful, though as discussed in chapter 9, took longer than expected. Despite a number of unexpected issues documented in section 9.4 I was able to perform experiments and generate meaningful data for analysis. The use of logging to record metrics was particular successful as it allowed me to go back to the logs at any point after and see exactly what happened in that experiment. The experiment logs are included on the CD submitted with this project. It would have been useful to have generated more test data that I did, as whilst it was certainly possible to extract useful knowledge from the data generated by these experiments, the level of certainty in the results could have been increased given a larger volume of data.

If continuing the work of this project it would be useful to perform experiments with a topology consisting of more operators. Under SR, replayed tuples have to travel from the source to the recovering operator whereas under R+SM replayed tuples have to travel only from preceding operator. As such I hypothesise that as the number of operators in the test topology increases SR would take longer to recover a failed operator whilst R+SM would recover the failed operator in relatively constant time.

Further work could also include experiments with real node failures, rather than simulated node failures. Such experiments would help to validate R+SM in Storm for industrial use.

Objective 4: Analyse and evaluate results and report on findings.

Using the results generated from the outcomes of objective 3 I was able to compare the performance of SR and R+SM in Storm and make a number of conclusions which are documented in sections 9.2 and 9.3.

I showed the R+SM can recover a failed operator more quickly than SR and with lower runtime overhead. Whilst the results show that R+SM is consistently faster than SR at recovering a failed operator, I expected the recovery time to increase under SR as the fail time within the processing window increased. I expected this due to the additional number of tuples that needed to be replayed and the extra distance replayed tuples have to travel under SR. This was shown in the results. As such further work would be useful to perform experiments on topologies with more operators, and with higher tuple input rates.

I also showed that the R+SM implementation presented here can cause significant and undesirable overhead after failure in some circumstances. Further work is required to identify the specific causes of such issues and modify the design to avoid them.

11. Evaluation, Reflections and Conclusions

This project has been extremely challenging and rewarding and I believe that I chose the correct project for me. Working with Apache Storm has taught me abount open-source software, Java, distributed systems and stream processing. I also learned about Clojure and functional programming though I did not use them on the project as much as I expected. This was both because I did not deem to be best choice for the implementation and also because the workload was already high, learning a new programming paradigm and language to a level suitable for the R+SM implementation was too much in combination with the rest of project. In the future this is an area I would like to continue learning about and practising. I also intend to continue enhancing my expertise in Stream processing as I find it interesting and believe there to be a growing industrial demand for this skill.

I have also improved my organisational and time management skills by running this large and ambitious project. In addition to my full-time job I have learned a large number of new skills, designed and implemented R+SM, designed and performed experiments and produced this report. This would not have been possible without dedication, motivation and good use of time and organisation.

The objectives of the project have been met as described in chapter 10. The objectives set for the project were the correct objectives in order to achieve the desired outcomes – implement and evaluate R+SM. The outcomes of the project have been successful, I have proven that R+SM can be implemented in Storm and contributed a basic design and implementation. This is of use to the Storm user and developer community. I have also contributed the validation of R+SM as an applicable and high performance fault-tolerance approach in Storm; this of use to the academic community.

I largely followed my project plan and performed the main iterations as planned. However designing and implementing the experiments for the experimental evaluation phase took considerably longer than I expected. It was much larger task than I realised in that more code was required and I had to diagnose issues and retrieve metrics measurements. I encountered issues running the experiments, these are described in section 9.4. Investigating and rectifying these issues took nearly 1 month and this caused a delay writing up the report.

If doing the project again I would still break the work into iterations as I did in my original project plan. However I would spread the experimental evaluation work across the iterations of design and development. So I would design and develop a small part of functionality and then perform experiments on it in a controlled environment. I would also automate the experiments and processing of metrics so that experiments could be repeated often and in a controlled way, without requiring manual intervention as my experiments did. Running the project in this way would allow me to use the results of experiments to drive further design and implementation changes and ensure that I was focusing my efforts on areas where I would achieve the most benefit in performance and also to identify issues early. This approach would also lead to a larger volume of test data generated in a controlled environment so more confidence could be placed on the results.

The unit tests that I performed did verify that the code was working as expected but since they were not run in a distributed environment with real failures, they were not able to identify the issues that I encountered when running experiments. Running the project with experiments at every iteration would avoid these issues.

At the beginning of the project I invested time learning the Clojure language and Storm. These activities were scheduled in the project plan. Learning Clojure and using this knowledge to study the Storm source code gave me a thorough grounding in how Storm works and informed my later design choices. However since I did not program the R+SM solution in Clojure, some of the time expended on these activities could potentially have been better used elsewhere on the project.

There were also areas of functionality that I spent time implementing but did not use for the experiments For example my intimal experiments had functionality to support sliding and tumbling processing windows. This was complex to implement and took time, however it was never used in the experiments. Having discovered the issues with latency and bolt re-instantiation I instead focused on a smaller set of experiments. This wasted time could have been used for other project activities, and would have been avoided if at each step I developed the minimum viable piece of R+SM implementation, then a minimum viable experiment to evaluate it; driving further changes based on the results of pervious experiments.

In hindsight I also think my design focused too much on being maintainable so that it could be extended to a full Storm implementation in future. I spent considerable time selecting the best design patterns to implement R+SM with. Whilst this made later stages of the project easier because the code was easier to work with and more flexible, producing a production ready implementation of R+SM was not a project objective. If doing the project again I would still strive to produce quality code but as mentioned several times already, I would drive the development using experiment results. For example I would code an aspect of R+SM in the simplest way possible, then perform automated experiments and evolve the design in order to improve the experiment results.

To summarize, in this project have I shown that R+SM can be applied in Apache Storm and can perform recovery from failure more quickly and with less overhead than SR. This makes a contribution to the body of academic work by validating the work of Castro Fernandez *et al.*, 2013. My findings are also of benefit to the Storm user community as R+SM is shown to perform better than SR and I have contributed a base implementation that could be extended for use within industry.

I recommend further investigation into R+SM in Storm. This would take the form of further development to make the implementation suitable for industrial use, and additional experimentation to exercise R+SM with additional workloads and failure scenarios.

12. Glossary

Term	Meaning
SR	Source replay
R+SM	Recovery using state management
IDE	Integrated development environment
SPS	Stream processing system
BBD	`BufferingBoltDecorator` class
RBD	`RecoverableBoltDecorator` class
BOC	`BufferingOutputCollector` class

13. References

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E. and others (2005) ‘The Design of the Borealis Stream Processing Engine.’, in *CIDR*, pp. 277–289.

Agrawal, D., Das, S. and El Abbadi, A. (2011) ‘Big data and cloud computing: current state and future opportunities’, in *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, pp. 530–533.

apache/storm (no date). Available at: https://github.com/apache/storm (Accessed: 22 December 2014).

Apache Storm (no date). Available at: http://hortonworks.com/hadoop/storm/ (Accessed: 23 December 2014).

Apache Thrift - Home (no date). Available at: https://thrift.apache.org/ (Accessed: 7 January 2015).

Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N. and others (2004) ‘Retrospective on aurora’, *The VLDB Journal*, 13(4), pp. 370–383.

Balazinska, M., Hwang, J.-H. and Shah, M. A. (2009) ‘Fault-tolerance and high availability in data stream management systems’, in *Encyclopedia of Database Systems*. Springer, pp. 1109–1115.

Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E. and Pietzuch, P. (2013) ‘Integrating scale out and fault tolerance in stream processing using operator state management’, in *Proceedings of the 2013 international conference on Management of data*. ACM, pp. 725–736.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y. and Zdonik, S. B. (2003) ‘Scalable Distributed Stream Processing.’, in *CIDR*, pp. 257–268.

Dawson, C. W. (2005) *Projects in computing and information systems: a student’s guide*. Pearson Education.

Dean, J. and Ghemawat, S. (2008) *MapReduce: simplified data processing on large clusters*. (1). Available at: www.summon.com.

Docker - Build, Ship, and Run Any App, Anywhere (no date). Available at: https://www.docker.com/ (Accessed: 8 January 2015).

docker/fig (no date). Available at: https://github.com/docker/fig (Accessed: 4 January 2015).

EsotericSoftware/kryo (no date). Available at: https://github.com/EsotericSoftware/kryo (Accessed: 29 December 2014).

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design patterns: elements of reusable object-oriented software*. Pearson Education.

GNU grep - GNU Grep: Print lines matching a pattern - GNU Project - Free Software Foundation (no date). Available at: https://www.gnu.org/software/grep/manual/ (Accessed: 5 January 2015).

Hunt, P., Konar, M., Junqueira, F. P. and Reed, B. (2010) ‘ZooKeeper: wait-free coordination for internet-scale systems’, in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pp. 11–11.

IntelliJ IDEA — The Most Intelligent Java IDE (no date). Available at: https://www.jetbrains.com/idea/ (Accessed: 8 January 2015).

Kraus, J. and Kestler, H. (2009) ‘Multi-core parallelization in Clojure: a case study’, in *Proceedings of the 6th European Lisp Workshop*. ACM, pp. 8–17. doi: 10.1145/1562868.1562870.

LMAX-Exchange/disruptor (no date). Available at: https://github.com/LMAX-Exchange/disruptor (Accessed: 22 December 2014).

Log4j 2 Guide - Apache Log4j 2 (no date). Available at: http://logging.apache.org/log4j/2.x/ (Accessed: 28 December 2014).

Mackinnon, T., Freeman, S. and Craig, P. (2001) ‘Endo-testing: unit testing with mock objects’, *Extreme programming examined*, pp. 287–301.

mockito/mockito (no date). Available at: https://github.com/mockito/mockito (Accessed: 8 January 2015).

Netty: Home (no date). Available at: http://netty.io/ (Accessed: 25 December 2014).

Neumeyer, L., Neumeyer, L., Robbins, B., Robbins, B., Nair, A., Nair, A., Kesari, A. and Kesari, A. (2010) ‘S4: Distributed Stream Computing Platform’, in. IEEE, pp. 170–177. Available at: www.summon.com.

Oates, B. J. (2006) *Researching information systems and computing*. London: Sage.

Powered By Storm (no date). Available at: http://storm.apache.org/documentation/Powered-By.html (Accessed: 22 December 2014).

Rationale (no date). Available at: http://storm.apache.org/documentation/Rationale.html (Accessed: 22 December 2014).

Redis (no date). Available at: http://redis.io/ (Accessed: 26 December 2014).

Storm, distributed and fault-tolerant realtime computation (no date). Available at: http://storm.apache.org/ (Accessed: 22 December 2014).

Structure of the Codebase (no date). Available at: https://storm.apache.org/documentation/Structure-of-the-codebase.html (Accessed: 23 December 2014).

tail invocation - GNU Coreutils (no date). Available at: http://www.gnu.org/software/coreutils/manual/html_node/tail-invocation.html (Accessed: 5 January 2015).

The leading OS for PC, tablet, phone and cloud | Ubuntu (no date). Available at: http://www.ubuntu.com/ (Accessed: 8 January 2015).

Thompson, M., Farley, D., Barker, M., Gee, P. and Stewart, A. (2011) ‘Disruptor: High Performance Alternative To Bounded Queues For Exchanging Data Between Concurrent Threads’, *technical paper, LMAX Exchange*. Available at: http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf.

Trident Tutorial (no date). Available at: https://storm.apache.org/documentation/Trident-tutorial.html (Accessed: 29 December 2014).

wurstmeister/storm-docker (no date). Available at: https://github.com/wurstmeister/storm-docker (Accessed: 8 January 2015).

Zhang, Q., Cheng, L. and Boutaba, R. (2010) ‘Cloud computing: state-of-the-art and research challenges’, *Journal of internet services and applications*, 1(1), pp. 7–18.

Zobrist, A. L. (1970) ‘A new hashing method with application for game playing’, *ICCA journal*, 13(2), pp. 69–73.

14. Appendices

14.1. Appendix A - Project Proposal

Project proposal: Implementing operator state checkpoint based fault-tolerance in Apache Storm

14.1.1. Introduction

Within the realm of big data, stream processing systems (SPSs) are designed to process continuous streams of data in real-time. They have been the subject of a considerable body of work and have progressed from centralised systems, to parallel systems capable of being distributed across hundreds of servers.

By making a large number of servers available as a utility, cloud computing could be well suited to accommodate these modern SPSs. However, given inevitable machine and network failures, SPSs must be able to recover efficiently.

Recovery using state management (R+SM) (Castro Fernandez *et al.*, 2013) is a fault-tolerance mechanism for SPSs where operator state is externalised and periodic state checkpoints are made by the SPS. Recovery from failure can be achieved by restoring backed up state to a standby node. This project will implement the R+SM mechanism into the Apache Storm SPS. It will also evaluate the effectiveness of this mechanism compared to Storm's existing the source replay (SR) approach to failure recovery. Storm has been selected over other open-source SPSs because of its commercial take-up and the fact that it focuses on fault-tolerance as a core feature.

The research question to be addressed by this project is:

How does implementation of R+SM affect failure recovery in the Apache Storm SPS?

14.1.1.1. Aims and objectives

The aim of the work is to implement and evaluate operator state checkpoint based fault tolerance in the Apache Storm stream processing system.

The objectives of the work are as follows, please refer to the Planning section for milestone dates:

- Search, review and critique existing approaches for fault-tolerance in distributed SPSs. This objective will be ongoing throughout the project, it shall be completed by milestone 4 – 'Project complete'
- Design and implement R+SM in Apache Storm. This objective shall be completed by milestone 2 – 'Recovery of a three bolt topology'
- Prepare and execute experiments to evaluate the efficiency of R+SM in Apache Storm. This objective shall be completed by milestone 3 – 'Experimental evaluation completed'

- Produce final written report. This objective shall be completed by milestone 4 – 'Project complete'

14.1.1.2. Research Products

The research products of the work will consist of:

- Design of R+SM in Apache Storm
- Implementation of the above
- Experimental evaluation of R+SM in Apache Storm
- Written report including critical context, design, methodology, results and evaluation

14.1.2. Critical context

Big data analysis is now an effective business strategy, which has led to the development of a multitude of processing frameworks (Agrawal, Das and El Abbadi, 2011, pp. 1–2). Complimenting batch processing systems such as MapReduce (Dean and Ghemawat, 2008), stream processing systems (SPS) provide real-time processing of continuous streams of data in a manner that can handle sudden increases in input volume (Balakrishnan *et al.*, 2004, p. 1).

SPSs have been the subject of considerable research (Abadi *et al.*, 2005, p. 1) progressing from centralised systems such as Aurora (Cherniack *et al.*, 2003) to distributed systems such as Borealis (Abadi *et al.*, 2005). The current crop of highly scalable SPSs such as SEEP (Castro Fernandez *et al.*, 2013), Apache S4 (Neumeyer *et al.*, 2010) and Apache Storm (*Storm, distributed and fault-tolerant realtime computation*, no date) are capable of a high level of parallelism and distribution (Castro Fernandez *et al.*, 2013, p. 1).

The large resource pools and on-demand model offered by cloud computing providers (Zhang, Cheng and Boutaba, 2010, p. 1) are well-suited for modern SPSs, however failures are common in cloud environments (Zhang, Cheng and Boutaba, 2010, p. 6) as such SPSs must be able to recover in an efficient manner (Castro Fernandez *et al.*, 2013, p. 1).

Various approaches for SPS fault-tolerance are presented in the literature. Approaches fall into two categories known as active and passive replication (Balazinska, Hwang and Shah, 2009). In active replication, a redundant secondary query is processed alongside the primary query, in the event of failure of a primary node, the system can failover to the secondary. The techniques within the passive replication category are passive standby and upstream backup. In passive standby, the system regularly checkpoints operator state to backup nodes. To recover from failure the backup is first brought up-to-date by replaying tuples not present in the checkpoint; it can then replace the failed node – a process known as query repair. Upstream backup is similar to passive standby; however there is no checkpoint step. To recover from failure, the backup is brought up-to-date only by replaying tuples held in the output queues of upstream operators. An additional approach is the source replay (SR) approach utilised by Apache Storm. This is a special case of upstream backup where tuples are kept in the output queue of the source and replayed in the event of failure. SR is discussed in more detail within the Apache Storm section below.

(Castro Fernandez *et al.*, 2013) consider a number of fault-tolerance strategies from the literature in their discussion of the recovery of stateful operators, which are facilitated in modern systems such as Apache S4 and Apache Storm. They argue that active replication is too resource intensive and that passive replication is less resource intensive but too slow to recover. They also argue that upstream backup and SR are not practical for the recovery of operators that have state based upon the whole of the processed stream.

(Castro Fernandez *et al.*, 2013) contribute R+SM. This approach defines a model for representing query state, consisting of the processing, output buffer and routing states of all operators in the query. Also defined are a set of operations for managing state and using it to recover from failures. To recover from an operator failure, the SPS can use the latest operator checkpoint to backup and restore operator state to a node from a pool of standby nodes. The node can then be brought up-to-date using the output buffer state of upstream operators.

They observe that for a simple stream query their approach yields faster recovery times than the upstream backup and SR approaches. They also evaluate the overhead of their approach by exploring processing latency, but not in comparison to other approaches. Such comparison would have been useful; especially since the latency of Storm's SR approach has not been explored in the literature.

14.1.2.1. Apache Storm

Storm is an Apache Incubator project that was originally developed by the company Backtype before they were acquired by Twitter who open-sourced it. Storm has been selected for implementation of operator state based recovery for several reasons:

- Storm considers fault-tolerance a core-feature (*Rationale*, no date)
- The project is in active use by over 50 companies (*Powered By Storm*, no date)
- There is an active open-source community around Storm (*apache/storm*, no date)

The system aims to provide a flexible programming model, applicable to a range of problems and to hide much of the incidental complexity associated with distributed, parallel and fault-tolerant stream processing; allowing users to concentrate instead on business logic.

The source code for Storm is freely available under the Apache License on GitHub (*apache/storm*, no date); including a comprehensive suite of tests. Implementation logic and tests are written in Clojure, a functional language for the JVM in the Lisp family of languages. All interfaces are written in Java and the project uses the Apache Thrift RPC framework to allow client usage by a wide range of languages.

Storm is designed to run as a distributed system on a cluster of machines, consisting of a master node running the 'Nimbus' service that manages a number or worker nodes each of which run a 'Supervisor' service. Nodes are coordinated by the Apache Zookeeper (Hunt *et al.*, 2010) distributed application coordination service.

Computations in Storm (known as topologies) are represented as a graph of operators (known as bolts) and stream sources (known as spouts). Bolts are user defined operators; they may perform any action and can contain arbitrary state. Spouts output a continuous stream of data items known as tuples, resulting in topologies being continuous – that is they only end when killed by the user. The links between nodes in a topology indicate how streams will be processed through the system, being processed by each operator in turn. The level of parallelism required for each node can be user-specified and this is then managed by Storm, an example of a topology with parallel execution of a bolt is shown in figure 1 below.

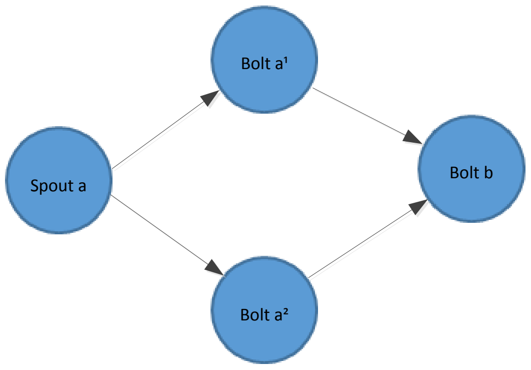


Figure 1 – topology with parallel execution

When a tuple emitted from a spout (a spout tuple) is processed by a bolt, that bolt may emit one or more new tuples, which when processed by downstream bolts, may result in further tuples being created. This results in a directed acyclic graph of tuples as show in figure 2. Fault-tolerance in Storm uses the SR model, which guarantees that every spout tuple will be processed. The user is required to inform the system whenever a new tuple is created (see figure 2 below) and when a tuple is processed; the system's 'Acker' service uses this information to track the complete processing of tuple graphs, replaying the tuple from the spout if the tuple graph is not marked as complete within a customisable time period. Storm uses a probabilistic hashing technique called Zobrist hashing (Zobrist, 1970) to monitor when tuples have been processed. In this technique a hash (known as an 'ack val') is associated with the ID of the spout tuple, this hash contains an XOR of the ids of all of the unprocessed tuples still in the graph. When the ack val reaches 0, the system knows the entire graph is processed and removes the spout tuple from the source.

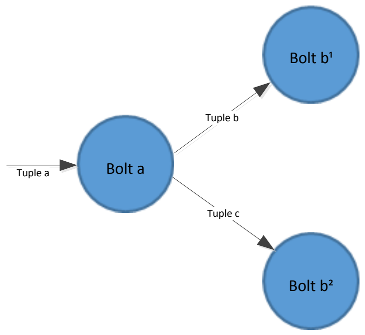


Figure 2 – partial topology, bolt a emits two tuples for each input tuple

(Castro Fernandez *et al.*, 2013, p. 2) argue that the SR approach ignores stateful bolts and that during recovery there is a negative effect on processing latency (p. 3). Their experiments show that SR does not recover a failed operator as quickly as R+SM but they do not evaluate the affect that recovery using SR has on processing latency. The nature of SR does support the claim that processing performance could be reduced during recovery, this is because in order to remain recoverable, a stateful bolt would be unable to mark as processed any tuple upon which its state depends. In the event of failure every unprocessed spout tuple would be replayed, which could be the entire history of spout tuples. Beyond the work of Castro Fernandez et al, it has not been possible to locate attention to the SR recovery approach within the academic literature. This project will contribute to the knowledge by evaluating the efficiency of the R+SM and SR recovery mechanisms in Storm.

14.1.3. Methods

The research strategy for the project will be a combination of the design and creation, and experimentation strategies (Briony J Oates, 2006), consisting of implementation of R+SM and experiments to evaluate the approach.

The requirements for the implementation are well-defined and the scope of the project is clear, especially given that there is no client. However some uncertainty stems from the fact that Storm is written in several languages and is a large and sophisticated system, that itself makes use of several other sophisticated systems such as Zookeeper and LMAX disrupter (*LMAX-Exchange/disruptor*, no date).

In light of this uncertainty the project will be controlled according to the prototyping software development software development lifecycle (SDLC) model. Prototyping has been selected due to its usefulness for “exploring technical uncertainties” (Dawson, 2005, p. 131) and in order to partition the work into a number of deliverables.

In the first iteration an experimental prototype will be built in order to explore implementation options and identify potentially problematic areas. The experimental prototype will inform later iterations. Once technical uncertainties are reduced, iterations two and three will take the form of evolutionary prototyping.

Iteration two will begin with the extraction of requirements from the (Castro Fernandez *et al.*, 2013) paper followed by design and then implementation stages. This iteration will feed into the design stage of iteration three. Since the requirements are clear and unlikely to change (given there is no client) the requirements stage of iteration three will not be required. At the end of iterations two and three, the work will be verified by recovering from the failure of two and three bolt stateful topologies respectively.

A review stage at the end of each iteration will be used to reassess risks and verify that the plan and methodologies are being followed and that expected progress is being made. The plan will be adjusted as appropriate to ensure that the project remains on course.

The top-down development method will be used for the project to ensure that new functionality integrates correctly with the existing system via system tests. The exact changes to be made and the order that they will be made in, will be planned as part of the first design phase, after experimental prototyping has revealed more about the solution. In line with the Storm’s existing architecture; implementation and test code will be written in a functional style using Clojure and interface code will be written in an object-orientated style using Java. Eclipse will be used as the IDE, since it supports both Clojure and Java. Git SCM will be used for source version control (backed up to a private online repository to ensure that no code is lost in the event of hard-drive failure or similar issue).

Storm is an active project and new releases are regular. Rather than continually merging new releases into the working copy of the project, a clone of the most recent stable version will be taken at the start of iteration one. Throughout the course of the project, releases by the Storm core team will be monitored, only those relevant to fault-tolerance will be merged into the project's working copy.

In addition to those tools already mentioned Grindstone will be used to track time spent on each task, Microsoft Project will be used to for producing Gantt charts and Zotero will be used for tracking references in the final report.

Following development and verification of the changes to Apache Storm, experiments will be designed to evaluate the hypothesis:

The R+SM recovery mechanism can be efficiently integrated in the Apache Storm SPS.

The measures of efficiency will be processing latency, tuple throughput and recovery time. Experiments will be carried out using simple topologies of two and three bolts. Results will be statistically analysed and graphically represented for evaluation.

14.1.4. Beneficiaries

The project will contribute to the body of knowledge by evaluating the applicability of R+SM as a viable strategy for failure recovery in a modern, distributed SPS with an active user base (*Powered By Storm*, no date) – it has previously only been implemented and evaluated within an experimental SPS.

The Apache Storm community will also benefit from the evaluation of R+SM as a potential alternative to Storm's existing SR recovery approach. Further, the implementation itself will be made publically available as a contribution for reference by the Storm open-source community.

14.1.5. Scope

The design and implementation of R+SM that will be delivered as part this project will be suitable for the evaluation of this approach in comparison to Apache Storm's existing recovery mechanism (SR). Preparing the implementation for integration into the core source code of Storm is not within scope.

Experimental evaluation will be carried out for two measures and using a simple test topology as detailed in the methods section below.

14.1.6. Resources

Eclipse and Apache Storm are open-source and freely available. Experiments will be run on 2 servers running Ubuntu Linux, each with a 4 core Intel Xeon E5-2407 2.20GHz processer and 8GB of RAM. These servers will be made available by the university.

14.1.7. Ethical issues

No human or animal participants are required for this project and the source-code for Apache Storm is available under the Apache 2 License, allowing personal use and modification. As such no ethical issues have been identified for the project. The completed research ethics checklist is at the end of this document.

14.1.8. Risks

Risks will be reassessed at the end of each iteration to identify any changes and plan appropriately.

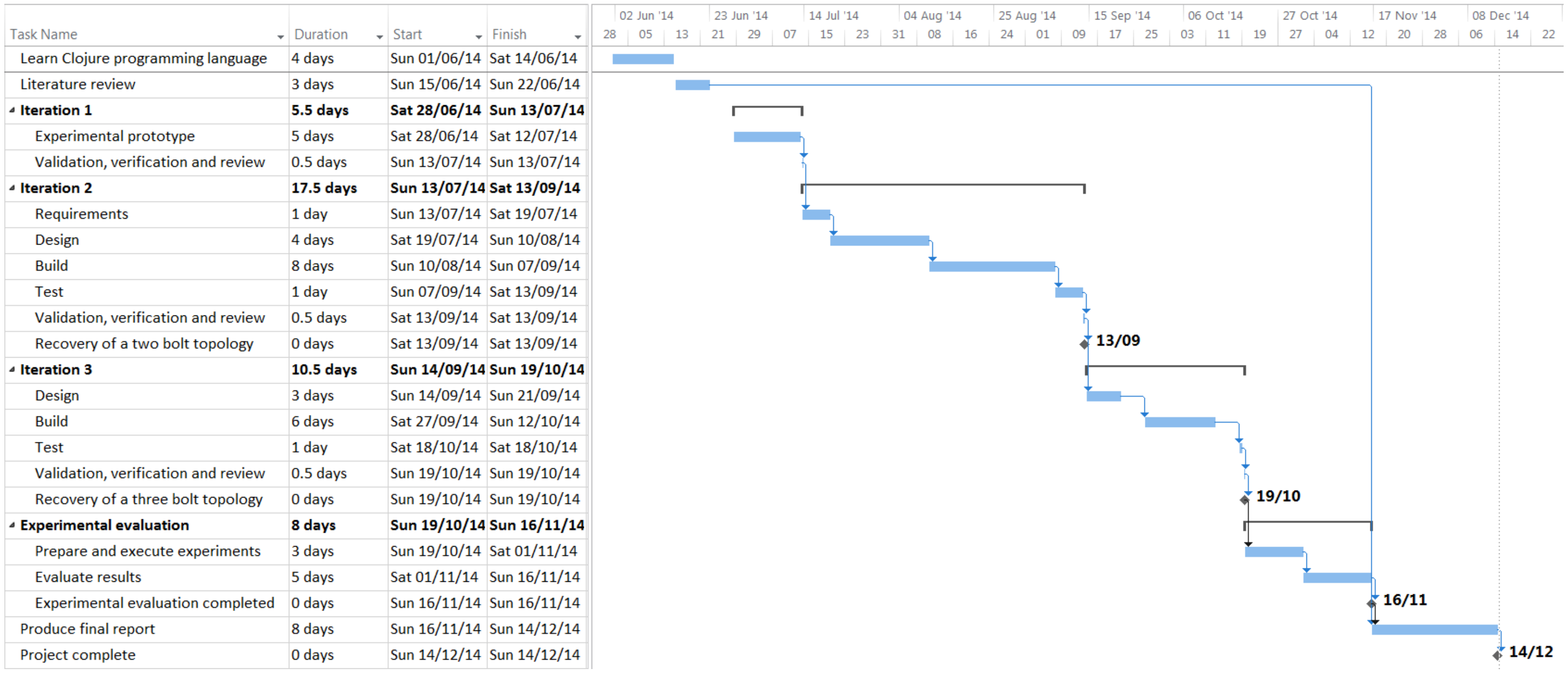
ID	Description	Likelihood (1-3)	Consequence (1-5)	Score	Mitigation
1	Employment obligations restrict time spent on project	1	3	3	Project plan schedules work for weekends. Add contingency to project plan. Save 10 days holiday as backup to catch-up lost time.
2	Supervisor no longer able to supervise project	1	3	3	A new supervisor will be allocated by university.
3	Illness	1	4	4	Add contingency to project plan. Save 10 days holiday as backup to catch-up lost time.
4	University test servers are unavailable	1	1	1	Pay for servers on Amazon EC2
5	Implementation too difficult	2	4	8	Experimental prototype and iterations will identify this issue early and the scope of the project can be adjusted accordingly.
6	Operator state checkpoint based fault-tolerance is implemented by the Storm core team	1	3	3	Evaluate Storm implementation vs source replay approach. No issues have been raised for this on Storm's issue list or develop communications.
7	Personal hardware/laptop etc. failure	2	1	2	All documents automatically backed-up on cloud storage (Dropbox) Source code backed up to cloud storage (BitBucket)
8	Difficulty learning Clojure programming	1	4	4	Appropriate time has been allocated in the project plan. Functional programming online course is being undertaken in advance of the project commencing.

14.1.8.1. Key

1-5	Low risk
6-10	Medium risk
11-20	High risk

14.1.9. Project plan

The project plan is based on working 16 hours a week (at weekends) and holidays are taken into account. Meetings with the project supervisor will take place bi-weekly. The project is scheduled to finish mid-December, providing some contingency.



Task Name	Duration	Start	Finish
Learn Clojure programming language	4 days	Sun 01/06/14	Sat 14/06/14
Literature review	3 days	Sun 15/06/14	Sun 22/06/14
Iteration 1	**5.5 days**	**Sat 28/06/14**	**Sun 13/07/14**
Experimental prototype	5 days	Sat 28/06/14	Sat 12/07/14
Validation, verification and review	0.5 days	Sun 13/07/14	Sun 13/07/14
Iteration 2	**17.5 days**	**Sun 13/07/14**	**Sat 13/09/14**
Requirements	1 day	Sun 13/07/14	Sat 19/07/14
Design	4 days	Sat 19/07/14	Sun 10/08/14
Build	8 days	Sun 10/08/14	Sun 07/09/14
Test	1 day	Sun 07/09/14	Sat 13/09/14
Validation, verification and review	0.5 days	Sat 13/09/14	Sat 13/09/14
Recovery of a two bolt topology	0 days	Sat 13/09/14	Sat 13/09/14
Iteration 3	**10.5 days**	**Sun 14/09/14**	**Sun 19/10/14**
Design	3 days	Sun 14/09/14	Sun 21/09/14
Build	6 days	Sat 27/09/14	Sun 12/10/14
Test	1 day	Sat 18/10/14	Sat 18/10/14
Validation, verification and review	0.5 days	Sun 19/10/14	Sun 19/10/14
Recovery of a three bolt topology	0 days	Sun 19/10/14	Sun 19/10/14
Experimental evaluation	**8 days**	**Sun 19/10/14**	**Sun 16/11/14**
Prepare and execute experiments	3 days	Sun 19/10/14	Sat 01/11/14
Evaluate results	5 days	Sat 01/11/14	Sun 16/11/14
Experimental evaluation completed	0 days	Sun 16/11/14	Sun 16/11/14
Produce final report	8 days	Sun 16/11/14	Sun 14/12/14
Project complete	0 days	Sun 14/12/14	Sun 14/12/14

14.1.10.References

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E. and others (2005) 'The Design of the Borealis Stream Processing Engine.', in *CIDR*, pp. 277–289.

Agrawal, D., Das, S. and El Abbadi, A. (2011) 'Big data and cloud computing: current state and future opportunities', in *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, pp. 530–533.

apache/incubator-storm (n.d.). Available at: https://github.com/apache/incubator-storm (Accessed: 11 April 2014).

Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Galvez, E., Salz, J., Stonebraker, M., Tatbul, N. and others (2004) 'Retrospective on aurora', *The VLDB Journal*, 13(4), pp. 370–383.

Balazinska, M., Hwang, J.-H. and Shah, M. A. (2009) 'Fault-tolerance and high availability in data stream management systems', in *Encyclopedia of Database Systems*. Springer, pp. 1109–1115.

Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E. and Pietzuch, P. (2013) 'Integrating scale out and fault tolerance in stream processing using operator state management', in *Proceedings of the 2013 international conference on Management of data*. ACM, pp. 725–736.

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y. and Zdonik, S. B. (2003) 'Scalable Distributed Stream Processing.', in *CIDR*, pp. 257–268.

Dawson, C. W. (2005) *Projects in computing and information systems: a student's guide*. Pearson Education.

Dean, J. and Ghemawat, S. (2008) *MapReduce: simplified data processing on large clusters*. (1). Available at: www.summon.com.

Hunt, P., Konar, M., Junqueira, F. P. and Reed, B. (2010) 'ZooKeeper: wait-free coordination for internet-scale systems', in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pp. 11–11.

LMAX-Exchange/disruptor (n.d.). Available at: https://github.com/LMAX-Exchange/disruptor (Accessed: 14 April 2014).

Neumeyer, L., Neumeyer, L., Robbins, B., Robbins, B., Nair, A., Nair, A., Kesari, A. and Kesari, A. (2010) 'S4: Distributed Stream Computing Platform', in. IEEE, pp. 170–177. Available at: www.summon.com.

Oates, B. J. (2006) *Researching information systems and computing*. London: Sage.

Powered By Storm (n.d.). Available at: http://storm.incubator.apache.org/documentation/Powered-By.html (Accessed: 11 April 2014).

Rationale (n.d.). Available at: http://storm.incubator.apache.org/documentation/Rationale.html (Accessed: 13 April 2014).

Storm, distributed and fault-tolerant realtime computation (n.d.). Available at: http://storm.incubator.apache.org/ (Accessed: 11 April 2014).

Zhang, Q., Cheng, L. and Boutaba, R. (2010) ‘Cloud computing: state-of-the-art and research challenges’, *Journal of internet services and applications*, 1(1), pp. 7–18.

Zobrist, A. L. (1970) ‘A new hashing method with application for game playing’, *ICCA journal*, 13(2), pp. 69–73.

14.1.11. Ethics checklist

	Research Ethics Checklist **School of Informatics BSc MSc/MA Projects**	
	If the answer to any of the following questions (1 – 3) is NO, your project needs to be modified.	*Delete as appropriate*
1	Does your project pose only minimal and predictable risk to you (the student)?	**Yes**
2	Does your project pose only minimal and predictable risk to other people affected by or participating in the project?	**Yes**
3	Is your project supervised by a member of academic staff of the School of Informatics or another individual approved by the module leaders?	**Yes**
	If the answer to either of the following questions (4 – 5) is YES, you MUST apply to the University Research Ethics Committee for approval. (You should seek advice about this from your project supervisor at an early stage.)	*Delete as appropriate*
4	Does your project involve animals?	**No**
5	Does your project involve pregnant women or women in labour?	**No**
	If the answer to the following question (6) is YES, you MUST complete the remainder of this form (7 – 19). If the answer is NO, you are finished.	*Delete as appropriate*
6	Does your project involve human participants? For example, as interviewees, respondents to a questionnaire or participants in evaluation or testing?	**No**
	If the answer to any of the following questions (7 – 13) is YES, you MUST apply to the Informatics Research Ethics Panel for approval and your application may be referred to the University Research Ethics Committee. (You should seek advice about this from your project supervisor at an early stage.)	*Delete as appropriate*
7	Could your project uncover illegal activities?	**N/A**
8	Could your project cause stress or anxiety in the participants?	**N/A**
9	Will you be asking questions of a sensitive nature?	**N/A**
10	Does your project rely on covert observation of the participants?	**N/A**
11	Does your project involve participants who are under the age of 18?	**N/A**
12	Does your project involve adults who are vulnerable because of their social, psychological or medical circumstances (vulnerable adults)?	**N/A**
13	Does your project involve participants who have learning difficulties?	**N/A**
	The following questions (14 – 16) must be answered YES, i.e. you MUST COMMIT to satisfy these conditions and have an appropriate plan to ensure they are satisfied.	*Delete as appropriate*

14	Will you ensure that participants taking part in your project are fully informed about the purpose of the research?	**N/A**
15	Will you ensure that participants taking part in your project are fully informed about the procedures affecting them or affecting any information collected about them, including information about how the data will be used, to whom it will be disclosed, and how long it will be kept?	**N/A**
16	When people agree to participate in your project, will it be made clear to them that they may withdraw (i.e. not participate) at any time without any penalty?	**N/A**
	The following questions (17 – 19) must be answered and the requested information provided.	*Delete as appropriate*
17	Will consent be obtained from the participants in your project? Consent from participants will be necessary if you plan to gather personal, medical or other sensitive data about them. "Personal data" means data relating to an identifiable living person; e.g. data you collect using questionnaires, observations, interviews, computer logs. The person might be identifiable if you record their name, username, student id, DNA, fingerprint, etc. *If **YES**, provide the consent request form that you will use and indicate who will obtain the consent, how are you intending to arrange for a copy of the signed consent form for the participants, when will they receive it and how long the participants will have between receiving information about the study and giving consent, and when the filled consent request forms will be available for inspection (**NOTE**: subsequent failure to provide the filled consent request forms will automatically result in withdrawal of any earlier ethical approval of your project):*	**N/A**
18	Have you made arrangements to ensure that material and/or private information obtained from or about the participating individuals will remain confidential? Provide details:	**N/A**
19	Will the research be conducted in the participant's home or other non-University location? If YES, provide details of how your safety will be preserved:	**N/A**

Templates

The templates available from the links below **must** be adapted according to the needs of your project before they are submitted for consideration. The sample form provided for projects involving children is to be used by the parents/guardians of the children participating in the research project.
Adult information sheet:

http://www.city.ac.uk/__data/assets/word_doc/0018/153441/TEMPLATE-FOR-PARTICIAPNT-INFORMATION-SHEET.doc

Adult consent form:

http://www.city.ac.uk/__data/assets/word_doc/0004/153418/TEMPLATE-FOR-CONSENT-FORM.doc

Child information sheet:

http://www.city.ac.uk/__data/assets/word_doc/0003/153462/Sample-Child-Information-Sheet.doc

Child consent form:

http://www.city.ac.uk/__data/assets/word_doc/0020/153461/Sample-child-consent-1.doc

14.2. Appendix B - Source code

Source code has also been supplied on a CD submitted with this report.

14.2.1. BufferingBoltDecorator.java

```
package ts.components;


import backtype.storm.task.IOutputCollector;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import org.joda.time.DateTime;
import ts.IHasBufferState;
import ts.TopologyUtilities;
import ts.TupleBuffer;
import org.apache.log4j.Logger;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Set;

class BufferingOutputCollector extends OutputCollector {
  TupleBuffer buffer;

  public BufferingOutputCollector(IOutputCollector delegate, TupleBuffer buffer) {
    super(delegate);
    this.buffer = buffer;
  }

  @Override
  public List<Integer> emit(List<Object> tuple) {
    DateTime timestamp = new DateTime();
    Values newTuple = new Values();
    newTuple.add(timestamp);
    newTuple.addAll(tuple);

    List<Integer> sentTo = emit(Utils.DEFAULT_STREAM_ID, newTuple);

    for(Integer t : sentTo) {
      buffer.add(t, timestamp, newTuple);
    }

    return sentTo;
  }
}

class WithTimestampDeclarer implements OutputFieldsDeclarer {
  private OutputFieldsDeclarer delegate;
  private Fields declaredFields;

  public WithTimestampDeclarer(OutputFieldsDeclarer delegate) {
    this.delegate = delegate;
  }

  public Fields getDeclaredFields() {
    return declaredFields;
  }

  @Override
  public void declare(Fields fields) {
    List<String> newFields = new ArrayList<String>();
    newFields.add("timestamp");
    newFields.addAll(fields.toList());
    declaredFields = new Fields(newFields);
    delegate.declare(declaredFields);
  }

  @Override
  public void declare(boolean direct, Fields fields) {
    delegate.declare(direct, fields);
  }

  @Override
  public void declareStream(String streamId, Fields fields) {
    delegate.declareStream(streamId, fields);
  }

  @Override
  public void declareStream(String streamId, boolean direct, Fields fields) {
    delegate.declareStream(streamId, direct, fields);

  }
}

public class BufferingBoltDecorator extends BaseRichBolt implements IHasBufferState {

  OutputCollector collector;
  TupleBuffer buffer;

  BaseRichBolt component;

  Logger log;
  int taskId;
```

```
    DateTime startupTime;

    public BufferingBoltDecorator(BaseRichBolt component) {
        this.component = component;
    }

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        buffer = new TupleBuffer(getDownstreamTaskIds(context));
        this.collector = new BufferingOutputCollector(collector, buffer);

        log = org.apache.log4j.Logger.getLogger(BufferingBoltDecorator.class);
        taskId = context.getThisTaskId();

        startupTime = new DateTime();

        component.prepare(stormConf, context, this.collector);

        log.info("TS: prepare called on buffering bolt decorator for task: " + taskId);
    }

    private List<Integer> getDownstreamTaskIds(TopologyContext context) {
        List<Integer> downstreamTaskIds = new ArrayList<Integer>();
        Set<String>                                    downStreamComponents                                    =
context.getThisTargets().get(Utils.DEFAULT_STREAM_ID).keySet();
        for(String downstreamComponent : downStreamComponents) {
            downstreamTaskIds.addAll(context.getComponentTasks(downstreamComponent));
        }
        return downstreamTaskIds;
    }

    @Override
    public void execute(Tuple tuple) {
        if(TopologyUtilities.isAckTuple(tuple)) {
            log.info(
                    "TS: received ack:" + tuple.getSourceComponent() +
                            ":" + tuple.getSourceTask() +
                            ", stream:" + tuple.getSourceStreamId() +
                            ", id:" + tuple.getMessageId() +
                            ", "  + tuple.getValues());

            buffer.trim(tuple.getSourceTask(), (DateTime) tuple.getValue(0));
        } else if(TopologyUtilities.isRecoveryRequestTuple(tuple)) {
            log.info(
                    "TS: received recover request:" + tuple.getSourceComponent() +
                            ":" + tuple.getSourceTask() +
                            ", stream:" + tuple.getSourceStreamId() +
                            ", id:" + tuple.getMessageId() +
                            ", "  + tuple.getValues());
            int taskToRecover = tuple.getSourceTask();
            List<Values> recoveryTuples = buffer.getTuplesForTask(taskToRecover);
            log.info("Split has " + recoveryTuples.size() + " buffered");

            if(startupTime.plusSeconds(5).isAfter(new DateTime())) {
              log.info("TS: recover request rejected. Topology starting up. Recorded startupTime is " +
startupTime);
              return;
            }

            if(recoveryTuples.size() == 0) {
              log.info("TS: Task " + tuple.getSourceTask() + " has no tuples to replay for task " +
taskToRecover);
            }

            for(Values recoveryTuple : recoveryTuples) {
              collector.emitDirect(taskToRecover,                    TopologyUtilities.RECOVERY_STREAM_ID,
recoveryTuple);
            }

            log.info("TS: Tuple replay to task " + taskToRecover + " completed by task " +
tuple.getSourceTask());
        }
        else {
            component.execute(tuple);
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        WithTimestampDeclarer withTimestampDeclarer = new WithTimestampDeclarer(declarer);
        component.declareOutputFields(withTimestampDeclarer);
        declarer.declareStream(TopologyUtilities.RECOVERY_STREAM_ID,
withTimestampDeclarer.getDeclaredFields());
    }

    public TupleBuffer getBufferState() {
        return buffer;
    }
}
```

14.2.2. RecoverableBoltDecorator.java

```
package ts.components;

import backtype.storm.Config;
import backtype.storm.generated.GlobalStreamId;
import backtype.storm.generated.Grouping;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import org.apache.log4j.Logger;
import org.joda.time.DateTime;
import redis.clients.jedis.Jedis;
import storm.starter.util.TupleHelpers;
import ts.IHasProcessingState;
import ts.ProcessingState;
import ts.TopologyUtilities;
import ts.experiments.wordcount.WindowActionTracker;
import ts.experiments.wordcount.checkpointed.CheckpointWindowedWordCountBolt;
import ts.util.StringSerialzer;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class RecoverableBoltDecorator extends BaseRichBolt {
  OutputCollector collector;
  TopologyContext context;
  transient Jedis jedis;
  HashMap<Integer, DateTime> ackInfo;
  WindowActionTracker ackTracker;
  BaseRichBolt component;
  ProcessingState restoredProcessingState;
  Logger log;

  public RecoverableBoltDecorator(BaseRichBolt component, int windowSecs, int ackAt) {
    this.component = component;
    this.ackTracker = new WindowActionTracker("Ack tracker", windowSecs, ackAt);
  }

  public RecoverableBoltDecorator(BaseRichBolt component, int ackAt, Jedis jedis) {
    this(component, 30, ackAt);
    this.jedis = jedis;
  }

  public RecoverableBoltDecorator(BaseRichBolt component, int ackAt) {
    this(component, 30, ackAt);
  }

  @Override
  public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
    this.context = context;
    ackInfo = new HashMap<Integer, DateTime>();

    this.collector = collector;

    log = org.apache.log4j.Logger.getLogger(RecoverableBoltDecorator.class);

    /*
    // This is how it would work in practice, but for the experiments, this done by the application
(decorated) bolt.
    restoreProcessingState(context.getThisTaskId());
    requestBufferState(context);
    */

    ((CheckpointWindowedWordCountBolt)component).prepare(stormConf, context, this.collector,
this);
  }

  public void restoreProcessingState(int taskId) {
    log.info("Task: " + context.getThisTaskId() + " requesting processing state");
    if(component instanceof IHasProcessingState) {
        String serializedState = getJedis().get(Integer.toString(taskId));

        if(serializedState == null) {
            // No processing state to retrieve
            return;
        }

        restoredProcessingState = (ProcessingState)StringSerialzer.deserialize(serializedState);
        ((IHasProcessingState)component).setProcessingState(restoredProcessingState);
    }
  }

  public void requestBufferState(TopologyContext context) {
    log.info("Task: " + context.getThisTaskId() + " requesting buffer state");
    ArrayList<Integer> taskIds = new ArrayList<Integer>();
    Map<GlobalStreamId, Grouping> sources = context.getThisSources();

    for(GlobalStreamId id : sources.keySet()) {
        if(id.get_streamId().equals(Utils.DEFAULT_STREAM_ID)) {
            String component = id.get_componentId();
            taskIds.addAll(context.getComponentTasks(component));
        }
    }
    for(Integer task : taskIds) {
        log.info("Requesting recovery from task: " + task);
        this.collector.emitDirect(task,   TopologyUtilities.REQUEST_BUFFER_STATE_STREAM_ID,   new
Values(TopologyUtilities.REQUEST_BUFFER_STATE_PAYLOAD));
    }
  }

  @Override
  public void execute(Tuple tuple) {
    if(TupleHelpers.isTickTuple(tuple)) {
        component.execute(tuple);

        ackTracker.update(log);
        if(ackTracker.shouldPerformRepeatableAction()) {
            checkpointState();
            sendAcks();
        }
        return;
    } else if(TopologyUtilities.isRecoveryTuple(tuple)) {
        if(restoredProcessingState != null && tupleReflectedInProcessingState(tuple)) {
            // Discard recovery tuple if already reflected in processing state
            return;
        }
    }

    component.execute(tuple);
    setAckInfo(tuple);
  }

  private boolean tupleReflectedInProcessingState(Tuple tuple) {
    DateTime tupleTimestamp = (DateTime)tuple.getValue(0);

    return tupleTimestamp.isBefore(restoredProcessingState.getTimestamp()) ||
        tupleTimestamp.isEqual(restoredProcessingState.getTimestamp());
  }

  private void checkpointState() {
    if(component instanceof IHasProcessingState) {
        ProcessingState processingState = ((IHasProcessingState)component).getProcessingState();
        String serialized = StringSerialzer.serialize(processingState);
        getJedis().set(Integer.toString(context.getThisTaskId()), serialized);
    }
  }

  private void setAckInfo(Tuple tuple) {
    ackInfo.put(tuple.getSourceTask(), (DateTime) tuple.getValue(0));
  }

  private void sendAcks() {
    for (Map.Entry<Integer, DateTime> entry : ackInfo.entrySet()) {
```

```
        Integer upstreamTaskId = entry.getKey();
        DateTime lastTupleTimestamp = entry.getValue();
        collector.emitDirect(upstreamTaskId,      TopologyUtilities.CHECKPOINT_STREAM_ID,      new
Values(lastTupleTimestamp));
      }
      ackInfo.clear();
   }

   @Override
   public void declareOutputFields(OutputFieldsDeclarer declarer) {
      component.declareOutputFields(declarer);
      declarer.declareStream(TopologyUtilities.CHECKPOINT_STREAM_ID, new Fields("tupleId"));
      declarer.declareStream(TopologyUtilities.REQUEST_BUFFER_STATE_STREAM_ID,              new
Fields("request"));
   }

   @Override
   public Map<String, Object> getComponentConfiguration() {
      Map<String, Object> conf = component.getComponentConfiguration();
      if(conf == null) {
         conf = new Config();
      }

      conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 1);
      return conf;
   }

   private Jedis getJedis() {
      if(jedis == null) {
         String redisHost = System.getenv("REDIS_PORT_6379_TCP_ADDR");
         if(redisHost == null) redisHost = "0.0.0.0";
         jedis = new Jedis(redisHost);
      }
      return jedis;
   }
}
```

14.2.3. WindowActionTracker.java

This class takes inspiration from Storm’s `SlidingWindowCounter` and `SlotBasedCounter` classes.

```
package ts.experiments.wordcount;

import org.apache.log4j.Logger;
import java.io.Serializable;

public class WindowActionTracker implements Serializable {

    private final int slots;
    private String name;
    private final int actionAt;
    private int current;

    public WindowActionTracker(String name, int slots, int actionAt) {
        this.name = name;
        this.actionAt = actionAt;
        this.slots = slots;
        this.current = 0;
    }

    public void update(Logger log) {
        current = (current % slots) + 1;
        if (current == 1) {
            log.info(name + " window start is now at slot 1");
        }
    }

    public boolean shouldPerformRepeatableAction() {
        return current != 0 && current % actionAt == 0 ;
    }

    public boolean shouldPerformOncePerWindowAction() {
        return current == actionAt;
    }
}
```

14.2.4. WordCounts.java

```
package ts.experiments.wordcount;

import java.io.Serializable;
import java.util.HashMap;
import java.util.Map;

public class WordCounts implements Serializable {
  private Map<Object, Long> counts;

  public WordCounts() {
    counts = new HashMap<Object, Long>();
  }

  public void incrementCountFor(Object word) {
    Long count = counts.get(word);
    if (count == null) {
      count = (long) 0;
    }
    count += 1;
    counts.put(word, count);
  }

  public Map<Object, Long> getCountsForAllWords() {
    return counts;
  }

  public void clearCounts() {
    counts = new HashMap<Object, Long>();
  }
}
```

14.2.5. CheckpointSentenceSpout.java

```
package ts.experiments.wordcount.checkpointed;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import org.apache.log4j.Logger;
import org.joda.time.DateTime;
import redis.clients.jedis.Jedis;

import java.util.Map;
import java.util.Random;

public class CheckpointSentenceSpout extends BaseRichSpout {
  public static final String FAIL_PAYLOAD = "TS-FAIL-PAYLOAD";

  private SpoutOutputCollector collector;
  private Random random;
  private DateTime lastSignalCheckTime;
  private int sleepMilliSecs = 0;
  private Logger log;
  private transient Jedis jedis;

  public CheckpointSentenceSpout(int sleepMilliSecs) {
    this.sleepMilliSecs = sleepMilliSecs;
  }

  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
    log = org.apache.log4j.Logger.getLogger(CheckpointSentenceSpout.class);
    random = new Random();
    lastSignalCheckTime = new DateTime();
  }

  String[] source = new String[]{
      "tinker taylor soldier spy",
      "the good the bad and the ugly",
      "from dusk till dawn",
      "alice in wonderland",
      "gone with the wind"
  };

  private String nextSentence() {
    return source[random.nextInt(source.length)];
  }
```

```
@Override
public void nextTuple() {
   if(sleepMilliSecs != 0) {
      Utils.sleep(sleepMilliSecs);
   }

   Object sentence;

   DateTime now = new DateTime();
   if(lastSignalCheckTime.plusSeconds(20).isBefore(now)) {
      Boolean shouldSendSignal = getJedis().exists("ts-start-test");
      if(shouldSendSignal) {
         getJedis().del("ts-start-test");
         collector.emit(new Values(FAIL_PAYLOAD, new DateTime()));
         return;
      }

      lastSignalCheckTime = now;
   }

   sentence = nextSentence();
   collector.emit(new Values(sentence, new DateTime()));
}

@Override
public void ack(Object id) {}

@Override
public void fail(Object id) {}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
   declarer.declare(new Fields("sentence", "created"));
}

private Jedis getJedis() {
   if(jedis == null) {
      String redisHost = System.getenv("REDIS_PORT_6379_TCP_ADDR");
      if(redisHost == null) redisHost = "0.0.0.0";
      jedis = new Jedis(redisHost);
   }
   return jedis;
}

@Override
public void activate() {
   log.info("Activate called on spout");
}
```

```
    @Override
    public void deactivate() {
        log.info("Deactivate called on spout");
    }
}
```

14.2.6. CheckpointSplitBolt.java

```
package ts.experiments.wordcount.checkpointed;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import org.joda.time.DateTime;

import java.util.Map;

public class CheckpointSplitBolt extends BaseRichBolt {
  OutputCollector collector;

  @Override
  public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
    this.collector = collector;
  }

  @Override
  public void execute(Tuple tuple) {
    String[] words = tuple.getString(0).split(" ");
    DateTime rootCreated = (DateTime)tuple.getValue(1);
    for(String word : words) {
      collector.emit(new Values(word, rootCreated));
    }
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("word", "created"));
  }
}
```

14.2.7.CheckpointWindowedWordCountBolt.java

```
package ts.experiments.wordcount.checkpointed;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import org.apache.log4j.Logger;
import org.joda.time.DateTime;
import storm.starter.util.TupleHelpers;
import ts.IHasProcessingState;
import ts.ProcessingState;
import ts.TopologyUtilities;
import ts.components.RecoverableBoltDecorator;
import ts.experiments.wordcount.WindowActionTracker;
import ts.experiments.wordcount.WordCounts;

import java.util.Map;

// Implementation originally influenced by storm.starter.bolt.RollingCountBolt.java by has diverged
significantly
public class CheckpointWindowedWordCountBolt extends BaseRichBolt implements
IHasProcessingState {

  private OutputCollector collector;
  private WordCounts wordCounts;
  private boolean shouldFail;
  private WindowActionTracker failTracker;
  private Logger log;
  private int taskId;
  private DateTime lastProcessed = new DateTime(0);
  private WindowActionTracker sendCountsTracker;
  private RecoverableBoltDecorator decorator;
  private TopologyContext context;

  public CheckpointWindowedWordCountBolt(int windowSecs, int failAtSlot) {
    this.failTracker = new WindowActionTracker("Fail tracker", windowSecs, failAtSlot);
    this.sendCountsTracker = new WindowActionTracker("Send counts tracker", windowSecs, 30);
    this.wordCounts = new WordCounts();
  }

  public void prepare(Map stormConf, TopologyContext context, OutputCollector collector,
RecoverableBoltDecorator decorator) {
    this.context = context;
    this.collector = collector;
    this.decorator = decorator;
```

```
        log = org.apache.log4j.Logger.getLogger(CheckpointWindowedWordCountBolt.class);
        taskId = context.getThisTaskId();
        log.info("Task: " + taskId + " preparing");
    }

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        throw new RuntimeException("Expect to be called with a RecoverableBoltDecorator for recovery
experiment");
    }

    @Override
    public void execute(Tuple input) {
        if(TupleHelpers.isTickTuple(input)) {
            failTracker.update(log);
            if(shouldFail && failTracker.shouldPerformOncePerWindowAction()) {
                shouldFail = false;
                log.warn("TS: Task: " + taskId + " failing");
                // Clear state
                this.wordCounts = new WordCounts();
                // Now recover
                log.warn("TS: Task: " + taskId + " starting recovery");
                decorator.restoreProcessingState(taskId);
                decorator.requestBufferState(context);
            }

            sendCountsTracker.update(log);
            if(sendCountsTracker.shouldPerformRepeatableAction()) {
                sendWordCounts();
            }
        } else {
            String word = input.getString(1);

            if (word.equals("TS-FAIL-PAYLOAD")) {
                // Don't want to fail if receive fail signal on the recovery stream
                if(TopologyUtilities.isRecoveryTuple(input)) {
                    log.warn("TS: Recovered signal received by task: " + taskId);
                } else {
                    log.warn("TS: Fail signal received by task: " + taskId);
                    shouldFail = true;
                }
            }

            wordCounts.incrementCountFor(word);

            setLastProcessedDate(input);

            DateTime rootCreated = (DateTime)input.getValue(2);
            long latency = new DateTime().getMillis() - rootCreated.getMillis();
            log.info("TS: tuple latency: " + latency);
        }
    }

    private void setLastProcessedDate(Tuple tuple) {
        lastProcessed = (DateTime)tuple.getValue(0);
    }

    private void sendWordCounts() {
        Map<Object, Long> counts = wordCounts.getCountsForAllWords();
        for (Map.Entry<Object, Long> entry : counts.entrySet()) {
            Object word = entry.getKey();
            Long count = entry.getValue();
            collector.emit(new Values(word, count));
        }
        log.info("Word counts emitted");
        wordCounts.clearCounts();
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }

    @Override
    public ProcessingState getProcessingState() {
        return new ProcessingState(lastProcessed, wordCounts);
    }

    @SuppressWarnings("unchecked")
    @Override
    public void setProcessingState(ProcessingState processingState) {
        lastProcessed = processingState.getTimestamp();
        wordCounts = (WordCounts) processingState.getState();
    }
}
```

14.2.8. CheckpointWordCountingTopology.java

```java
package ts.experiments.wordcount.checkpointed;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.metric.LoggingMetricsConsumer;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import ts.TopologyUtilities;
import ts.components.BufferingBoltDecorator;
import ts.components.RecoverableBoltDecorator;

public class CheckpointWordCountingTopology {
  public static void main(String[] args) throws Exception {

    final int windowSizeSecs = 30;
    final int ackFrequency = 5;
    final int failAtSlot = 14;
    final int pauseForMilliSecs = 100;


    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("spout", new CheckpointSentenceSpout(pauseForMilliSecs), 1)
        .setNumTasks(1);
    builder.setBolt("split", new BufferingBoltDecorator(new CheckpointSplitBolt()), 1)
        .setNumTasks(1)
        .shuffleGrouping("spout")
        .directGrouping("count", TopologyUtilities.CHECKPOINT_STREAM_ID)
        .directGrouping("count", TopologyUtilities.REQUEST_BUFFER_STATE_STREAM_ID);
    builder.setBolt("count",                    new                    RecoverableBoltDecorator(new
CheckpointWindowedWordCountBolt(windowSizeSecs, failAtSlot), windowSizeSecs, ackFrequency),
1)
        .setNumTasks(1)
        .fieldsGrouping("split", new Fields("word"))
        .directGrouping("split", TopologyUtilities.RECOVERY_STREAM_ID);

    Config conf = new Config();
    conf.setDebug(true);
    conf.setNumAckers(0);
    conf.registerMetricsConsumer(LoggingMetricsConsumer.class, 1);

    if (args != null && args.length > 0) {
      conf.setNumWorkers(6);

      StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
    }
    else {
            conf.setMaxTaskParallelism(4);

            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("storm-word-count", conf, builder.createTopology());

            Thread.sleep(10000);
            cluster.shutdown();
        }
    }
}
```

14.2.9.StormSentenceSpout.java

```
package ts.experiments.wordcount.storm;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import org.apache.log4j.Logger;
import org.joda.time.DateTime;
import redis.clients.jedis.Jedis;
import ts.TopologyUtilities;

import java.util.*;
import java.util.concurrent.atomic.AtomicLong;

public class StormSentenceSpout extends BaseRichSpout {
  public static final String FAIL_PAYLOAD = "TS-FAIL-PAYLOAD";

  private static AtomicLong currentId;

  private SpoutOutputCollector collector;
  private Random random;
  private Map<Object, Object> pending;
  private Queue<Object[]> retry;
  private DateTime lastSignalCheckTime;
  private int sleepMilliSecs = 0;

  private Logger log;

  private transient Jedis jedis;

  public StormSentenceSpout() {}

  public StormSentenceSpout(int sleepMilliSecs) {
    this.sleepMilliSecs = sleepMilliSecs;
  }

  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
    log = org.apache.log4j.Logger.getLogger(StormSentenceSpout.class);
    random = new Random();
    currentId = new AtomicLong();
    pending = new LinkedHashMap<Object, Object>();
    retry = new LinkedList<Object[]>();
    lastSignalCheckTime = new DateTime();
```

```
}

String[] source = new String[]{
   "tinker taylor soldier spy",
   "the good the bad and the ugly",
   "from dusk till dawn",
   "alice in wonderland",
   "gone with the wind"
};

private String nextSentence() {
   return source[random.nextInt(source.length)];
}

private long nextMessageId() {
   return currentId.getAndIncrement();
}

@Override
public void nextTuple() {
   if(sleepMilliSecs != 0) {
      Utils.sleep(sleepMilliSecs);
   }

   Object id;
   Object sentence;

   DateTime now = new DateTime();
   if(lastSignalCheckTime.plusSeconds(20).isBefore(now)) {
      Boolean shouldSendSignal = getJedis().exists("ts-start-test");
      if(shouldSendSignal) {
         getJedis().del("ts-start-test");
         collector.emit(new Values(FAIL_PAYLOAD, new DateTime()));
         return;
      }

      lastSignalCheckTime = now;
   }

   if (retry.isEmpty()) {
      id = nextMessageId();
      sentence = nextSentence();
      pending.put(id, sentence);
      collector.emit(new Values(sentence, new DateTime()), id);
   }
   else {
      Object[] tupleToRetry = retry.remove();
      id = tupleToRetry[0];
      sentence = (tupleToRetry[1]);
```

```
            if(retry.isEmpty()) {
                log.info("TS: Retry queue is now empty");
            }

            pending.put(id, sentence);
            collector.emit(TopologyUtilities.RECOVERY_STREAM_ID,      new      Values(sentence,      new
DateTime()), id);
        }
    }

    @Override
    public void ack(Object id) {
        pending.remove(id);
    }

    @Override
    public void fail(Object id) {
        Object sentence = pending.remove(id);
        retry.add(new Object[]{id, sentence});
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        Fields fields = new Fields("sentence", "created");
        declarer.declare(fields);
        declarer.declareStream(TopologyUtilities.RECOVERY_STREAM_ID, fields);
    }

    private Jedis getJedis() {
        if(jedis == null) {
            String redisHost = System.getenv("REDIS_PORT_6379_TCP_ADDR");
            if(redisHost == null) redisHost = "0.0.0.0";
            jedis = new Jedis(redisHost);
        }
        return jedis;
    }

    @Override
    public void activate() {
        log.info("Activate called on spout");
    }

    @Override
    public void deactivate() {
        log.info("Deactivate called on spout");
    }
}
```

14.2.10.StormSplitBolt.java

```
package ts.experiments.wordcount.storm;

import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import org.joda.time.DateTime;
import ts.TopologyUtilities;

public class StormSplitBolt extends BaseBasicBolt {

  @Override
  public void execute(Tuple input, BasicOutputCollector collector) {
    String[] words = input.getString(0).split(" ");
    DateTime rootCreated = (DateTime)input.getValue(1);
    for(String word : words) {
      collector.emit(input.getSourceStreamId(), new Values(word, rootCreated));
    }
  }

  @Override
  public void cleanup() {

  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    Fields fields = new Fields("word", "created");
    declarer.declare(fields);
    declarer.declareStream(TopologyUtilities.RECOVERY_STREAM_ID, fields);
  }

}
```

14.2.11.StormWindowedWordCountBolt.java

```
package ts.experiments.wordcount.storm;

import backtype.storm.Config;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import org.apache.log4j.Logger;
import org.joda.time.DateTime;
import storm.starter.util.TupleHelpers;
import ts.TopologyUtilities;
import ts.experiments.wordcount.WindowActionTracker;
import ts.experiments.wordcount.WordCounts;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class StormWindowedWordCountBolt extends BaseRichBolt {

  private OutputCollector collector;
  private WordCounts wordCounts;
  private boolean shouldFail;
  private WindowActionTracker failTracker;
  private List<Tuple> pendingAck = new ArrayList<Tuple>();
  private Logger log;
  private int taskId;
  private WindowActionTracker sendCountsTracker;

  public StormWindowedWordCountBolt(int windowSecs, int failAtSlot) {
    this.failTracker = new WindowActionTracker("Fail tracker", windowSecs, failAtSlot);
    this.sendCountsTracker = new WindowActionTracker("Send counts tracker", windowSecs, 30);
    wordCounts = new WordCounts();
    pendingAck = new ArrayList<Tuple>();
  }

  @Override
  public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
    this.collector = collector;
    log = org.apache.log4j.Logger.getLogger(StormWindowedWordCountBolt.class);
    taskId = context.getThisTaskId();
    log.info("TS: Task: " + taskId + " preparing");
  }

  @Override
```

```
public void execute(Tuple input) {
   if(TupleHelpers.isTickTuple(input)) {
      failTracker.update(log);
      if(shouldFail && failTracker.shouldPerformOncePerWindowAction()) {
         shouldFail = false;
         log.warn("TS: Task: " + taskId + " failing");
         // Clear state and pending tuples
         this.wordCounts = new WordCounts();
         this.pendingAck = new ArrayList<Tuple>();
         // Now wait for recovery
         log.warn("TS: Task: " + taskId + " starting recovery");
      }

      sendCountsTracker.update(log);
      if(sendCountsTracker.shouldPerformRepeatableAction()) {
         sendWordCounts();
         sendAcks();
      }
   } else {
      String word = input.getString(0);

      if (word.equals("TS-FAIL-PAYLOAD")) {
         // Don't want to fail if receive fail signal on the recovery stream
         if(TopologyUtilities.isRecoveryTuple(input)) {
            log.warn("TS: Recovered signal received by task: " + taskId);
         } else {
            log.warn("TS: Fail signal received by task: " + taskId);
            shouldFail = true;
         }
      }


      wordCounts.incrementCountFor(word);
      pendingAck.add(input);

      DateTime rootCreated = (DateTime)input.getValue(1);
      long latency = new DateTime().getMillis() - rootCreated.getMillis();
      log.info("TS: tuple latency: " + latency);
   }
}

private void sendWordCounts() {
   Map<Object, Long> counts = wordCounts.getCountsForAllWords();
   for (Map.Entry<Object, Long> entry : counts.entrySet()) {
      Object word = entry.getKey();
      Long count = entry.getValue();
      collector.emit(new Values(word, count));
   }
   log.info("Word counts emitted");
   wordCounts.clearCounts();
```

```
    }

    private void sendAcks() {
        log.warn("TS: Task: " + taskId + " sending acks");
        for(Tuple t : pendingAck) {
            collector.ack(t);
        }
        pendingAck.clear();
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
        Map<String, Object> conf = new HashMap<String, Object>();
        conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, 1);
        return conf;
    }
}
```

14.2.12.StormWordCountingTopology.java

```
package ts.experiments.wordcount.storm;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.metric.LoggingMetricsConsumer;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import ts.TopologyUtilities;

public class StormWordCountingTopology {

   public static void main(String[] args) throws Exception {

      final int windowSizeSecs = 30;
      final int failAtSlot = 14;
      final int pauseForMilliSecs = 100;

      TopologyBuilder builder = new TopologyBuilder();

      builder.setSpout("spout", new StormSentenceSpout(pauseForMilliSecs), 1)
            .setNumTasks(1);
      builder.setBolt("split", new StormSplitBolt(), 1)
            .setNumTasks(1)
            .shuffleGrouping("spout")
            .shuffleGrouping("spout", TopologyUtilities.RECOVERY_STREAM_ID);
      builder.setBolt("count", new StormWindowedWordCountBolt(windowSizeSecs, failAtSlot), 1)
            .setNumTasks(1)
            .fieldsGrouping("split", new Fields("word"))
            .fieldsGrouping("split", TopologyUtilities.RECOVERY_STREAM_ID, new Fields("word"));

      Config conf = new Config();
      conf.setDebug(true);
      conf.setNumAckers(1);
      conf.registerMetricsConsumer(LoggingMetricsConsumer.class, 1);

      if (args != null && args.length > 0) {
         conf.setNumWorkers(6);

         StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
      }
      else {
         conf.setMaxTaskParallelism(4);

         LocalCluster cluster = new LocalCluster();
         cluster.submitTopology("storm-word-count", conf, builder.createTopology());

         Thread.sleep(10000);
         cluster.shutdown();
```

```
        }
    }
}
```

14.2.13.StringSerialzer.java

```
package ts.util;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class StringSerialzer {
   public static String serialize(Object o) {
      try {
         ByteArrayOutputStream bo = new ByteArrayOutputStream();
         ObjectOutputStream so = new ObjectOutputStream(bo);
         so.writeObject(o);
         so.flush();
         return bo.toString("ISO-8859-1");
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
   }

   public static Object deserialize(String s) {
      try {
         byte b[] = s.getBytes("ISO-8859-1");
         ByteArrayInputStream bi = new ByteArrayInputStream(b);
         ObjectInputStream si = new ObjectInputStream(bi);
         return si.readObject();
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
   }
}
```

14.2.14.IBufferState.java

```
package ts;

import backtype.storm.tuple.Values;

import java.util.Collection;

public interface IBufferState {
   public Collection<Values> getTuplesForTask(int id);
}
```

14.2.15.IHasBufferState.java

```
package ts;

public interface IHasBufferState {
   public IBufferState getBufferState();
}
```

14.2.16.IHasProcessingState.java

```
package ts;

public interface IHasProcessingState {
   public ProcessingState getProcessingState();

   public void setProcessingState(ProcessingState processingState);
}
```

14.2.17.ProcessingState.java

```
package ts;

import org.joda.time.DateTime;

import java.io.Serializable;

public class ProcessingState implements Serializable {

  private DateTime timestamp;

  private Object state;

  public ProcessingState(DateTime timestamp, Object state) {
    this.timestamp = timestamp;
    this.state = state;
  }

  public DateTime getTimestamp() {
    return timestamp;
  }

  public Object getState() {
    return state;
  }
}
```

14.2.18.TopologyUtilities.java

```
package ts;

import backtype.storm.tuple.Tuple;

public class TopologyUtilities {
   public static boolean isAckTuple(Tuple tuple) {
      return tuple.getSourceStreamId().equals(CHECKPOINT_STREAM_ID);
   }

   public static boolean isRecoveryRequestTuple(Tuple tuple) {
      return tuple.getSourceStreamId().equals(REQUEST_BUFFER_STATE_STREAM_ID);
   }

   public static boolean isRecoveryTuple(Tuple tuple) {
      return tuple.getSourceStreamId().equals(RECOVERY_STREAM_ID);
   }

   public static final String REQUEST_BUFFER_STATE_PAYLOAD = "request_buffer_state_payload";

   public static final String REQUEST_BUFFER_STATE_STREAM_ID = "ts_request_buffer_stream";

   public static final String RECOVERY_STREAM_ID = "ts_recovery_stream";

   public static final String CHECKPOINT_STREAM_ID = "ts_checkpoint_stream";
}
```

14.2.19.TupleBuffer.java

```
package ts;

import backtype.storm.tuple.Values;
import com.esotericsoftware.kryo.util.IntMap;
import org.joda.time.DateTime;

import java.util.*;

public class TupleBuffer implements IBufferState {
   IntMap<TreeMap<DateTime, List<Values>>> buffer;

   public TupleBuffer(List<Integer> downstreamTasks) {
//       buffer = new IntMap<TreeMap<DateTime, Values>>(downstreamTasks.size());
      buffer = new IntMap<TreeMap<DateTime, List<Values>>>(downstreamTasks.size());

      for(int t : downstreamTasks) {
         buffer.put(t, new TreeMap<DateTime, List<Values>>());
      }
   }

   public void add(int downstreamTaskId, DateTime timestamp, Values tupleValues) {
      TreeMap<DateTime, List<Values>> forTask = buffer.get(downstreamTaskId);
      List<Values> forTimestamp = forTask.get(timestamp);
      if(forTimestamp == null) {
         forTimestamp = new ArrayList<Values>();
      }
      forTimestamp.add(tupleValues);
      buffer.get(downstreamTaskId).put(timestamp, forTimestamp);
   }

   public void trim(int downstreamTaskId, DateTime to) {
      SortedMap<DateTime,  List<Values>>  toKeep  =  this.buffer.get(downstreamTaskId).tailMap(to,
false);
      this.buffer.put(downstreamTaskId, new TreeMap<DateTime, List<Values>>(toKeep));
   }

   @Override
   public List<Values> getTuplesForTask(int id) {
      List<Values> tuples = new ArrayList<Values>();
      Collection<List<Values>> allTuples = buffer.get(id).values();

      for(List<Values> valuesList : allTuples) {
         tuples.addAll(valuesList);
      }

      return tuples;
   }
}
```

14.2.20. BufferingBoltDecoratorTest.java

```
package ts;

import backtype.storm.generated.Grouping;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import org.joda.time.DateTime;
import org.mockito.ArgumentCaptor;
import org.testng.annotations.Test;
import storm.starter.tools.MockTupleHelpers;
import ts.components.BufferingBoltDecorator;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

import static org.fest.assertions.api.Assertions.assertThat;
import static org.mockito.Matchers.any;
import static org.mockito.Mockito.*;

public class BufferingBoltDecoratorTest {

  private Tuple mockNormalTuple() {
    return                                  MockTupleHelpers.mockTuple("someUpstreamComponent",
Utils.DEFAULT_STREAM_ID);
  }

  private Tuple mockAckTuple(DateTime timestamp) {
    Tuple          tuple          =          MockTupleHelpers.mockTuple("someDownstreamComponent",
TopologyUtilities.CHECKPOINT_STREAM_ID);
    when(tuple.getSourceTask()).thenReturn(1);
    when(tuple.getValue(0)).thenReturn(timestamp);
    return tuple;
  }

  private Tuple mockRequestRecoveryTuple() {
    Tuple          tuple          =          MockTupleHelpers.mockTuple("someDownstreamComponent",
TopologyUtilities.REQUEST_BUFFER_STATE_STREAM_ID);
    when(tuple.getSourceTask()).thenReturn(1);
    return tuple;
  }

  final Values NORMAL_TUPLE_PAYLOAD = new Values("test");
  private class MockComponent extends BaseRichBolt {
    OutputCollector collector;

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
      this.collector = collector;
    }

    @Override
    public void execute(Tuple tuple) {
      collector.emit(NORMAL_TUPLE_PAYLOAD);
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("some_string"));
    }
  }

  @SuppressWarnings("unchecked")
  private TopologyContext getMockContext() {
    TopologyContext context = mock(TopologyContext.class);

    HashMap<String, Map<String, Grouping>> mockTargets = (HashMap<String, Map<String,
Grouping>>)mock(HashMap.class);
    Map<String, Grouping> mockTargetGrouping = new HashMap<String, Grouping>();
    mockTargetGrouping.put("testComponent", null);
    when(mockTargets.get(Utils.DEFAULT_STREAM_ID)).thenReturn(mockTargetGrouping);

    when(context.getThisTargets()).thenReturn(mockTargets);

    when(context.getComponentTasks("testComponent")).thenReturn(new
ArrayList<Integer>(Arrays.asList(1)));
    return context;
  }

  @Test
  public void shouldPassNormalTupleToDecoratedComponent() {
    // given
    BaseRichBolt component = mock(BaseRichBolt.class);
    BufferingBoltDecorator bolt = new BufferingBoltDecorator(component);
    Map conf = mock(Map.class);
    TopologyContext context = getMockContext();
    OutputCollector collector = mock(OutputCollector.class);
    bolt.prepare(conf, context, collector);
    Tuple tuple = MockTupleHelpers.mockTuple("someUpstreamComponent",
Utils.DEFAULT_STREAM_ID);

    // when
```

```
        bolt.execute(tuple);

        // then
        verify(component).execute(tuple);
    }

    @Test
    public void shouldBufferEmittedTuples() {
        // given
        BaseRichBolt component = new MockComponent();
        BufferingBoltDecorator bolt = new BufferingBoltDecorator(component);
        Map conf = mock(Map.class);
        TopologyContext context = getMockContext();
        OutputCollector collector = mock(OutputCollector.class);
        when(collector.emit(eq(Utils.DEFAULT_STREAM_ID),                    anyCollectionOf(Tuple.class),
any(Values.class))).thenReturn(new ArrayList<Integer>(Arrays.asList(1)));
        bolt.prepare(conf, context, collector);
        Tuple         tuple         =         MockTupleHelpers.mockTuple("someUpstreamComponent",
Utils.DEFAULT_STREAM_ID);

        // when
        bolt.execute(tuple);

        // then
        assertThat(bolt.getBufferState().getTuplesForTask(1).size()).isEqualTo(1);
        assertThat(bolt.getBufferState().getTuplesForTask(1).get(0)).isInstanceOf(Values.class);
    }

    @SuppressWarnings("rawtypes")
    @Test
    public void shouldTrimTuples() {
        // given
        BaseRichBolt component = new MockComponent();
        BufferingBoltDecorator bolt = new BufferingBoltDecorator(component);
        Map conf = mock(Map.class);
        TopologyContext context = getMockContext();
        OutputCollector collector = mock(OutputCollector.class);
        when(collector.emit(
            eq(Utils.DEFAULT_STREAM_ID),
            anyCollectionOf(Tuple.class),
            any(Values.class))).thenReturn(new ArrayList<Integer>(Arrays.asList(1)));
        bolt.prepare(conf, context, collector);
        Tuple         tuple         =         MockTupleHelpers.mockTuple("someUpstreamComponent",
Utils.DEFAULT_STREAM_ID);

        // when
        bolt.execute(tuple);
        assertThat(bolt.getBufferState().getTuplesForTask(1).size()).isEqualTo(1);
        bolt.execute(mockAckTuple(new DateTime()));
        // then
        assertThat(bolt.getBufferState().getTuplesForTask(1).size()).isEqualTo(0);
    }

    @Test
    public void shouldReleaseTuplesWhenRequested() {
        // given
        BaseRichBolt component = new MockComponent();
        BufferingBoltDecorator bolt = new BufferingBoltDecorator(component);
        Map conf = mock(Map.class);
        TopologyContext context = getMockContext();
        OutputCollector collector = mock(OutputCollector.class);
        when(collector.emit(eq(Utils.DEFAULT_STREAM_ID),                    anyCollectionOf(Tuple.class),
any(Values.class)))
            .thenReturn(new ArrayList<Integer>(Arrays.asList(1)));
        bolt.prepare(conf, context, collector);
        // Add a buffered tuple
        bolt.execute(mockNormalTuple());

        // Needed because there is a check in the buffering bolt decorator to ensure it doesn't replay
when starting up
        Utils.sleep(5000);

        // when
        bolt.execute(mockRequestRecoveryTuple());

        // then
        ArgumentCaptor<Values> payloadArg = ArgumentCaptor.forClass(Values.class);

        verify(collector, times(1))
                .emitDirect(
                        eq(1),
                        eq(TopologyUtilities.RECOVERY_STREAM_ID),
                        anyCollectionOf(Tuple.class),
                        payloadArg.capture());

        DateTime                                        bufferedTupleDate                                        =
(DateTime)bolt.getBufferState().getTuplesForTask(1).get(0).get(0);
        Values payload = payloadArg.getAllValues().get(0);
        assertThat(payload.get(0)).isEqualTo(bufferedTupleDate);
        assertThat(payload.get(1)).isEqualTo("test");
    }
}
```

14.2.21.RecoverableBoltDecoratorTest.java

```
package ts;

import backtype.storm.generated.GlobalStreamId;
import backtype.storm.generated.Grouping;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import org.joda.time.DateTime;
import org.testng.annotations.Test;
import redis.clients.jedis.Jedis;
import storm.starter.tools.MockTupleHelpers;
import ts.components.RecoverableBoltDecorator;
import ts.util.StringSerialzer;

import java.util.*;

import static org.mockito.Mockito.*;

public class RecoverableBoltDecoratorTest {

  private Tuple mockNormalTuple(Object obj) {
    Tuple tuple = MockTupleHelpers.mockTuple("someUpstreamComponent",
Utils.DEFAULT_STREAM_ID);
    when(tuple.getValue(0)).thenReturn(obj);
    when(tuple.getSourceTask()).thenReturn(1);
    return tuple;
  }

  private Tuple mockRecoveryTuple(DateTime timestamp, Object obj) {
    Tuple tuple = MockTupleHelpers.mockTuple("someUpstreamComponent",
TopologyUtilities.RECOVERY_STREAM_ID);
    when(tuple.getValue(0)).thenReturn(timestamp);
    when(tuple.getValue(1)).thenReturn(obj);
    when(tuple.getSourceTask()).thenReturn(1);
    return tuple;
  }

  @Test
  public void shouldSendAckWithLatestTupleTimestamp() {
    // given
    BaseRichBolt component = mock(BaseRichBolt.class);
    RecoverableBoltDecorator bolt = new RecoverableBoltDecorator(component, 1);
    Map conf = mock(Map.class);
    TopologyContext context = mock(TopologyContext.class);
    OutputCollector collector = mock(OutputCollector.class);
    bolt.prepare(conf, context, collector);
```

```
        DateTime latestTimestamp = (new DateTime()).plusMinutes(1);
        bolt.execute(mockNormalTuple(new DateTime()));
        bolt.execute(mockNormalTuple(latestTimestamp));

        // when
        bolt.execute(MockTupleHelpers.mockTickTuple());

        // then
        verify(collector).emitDirect(1,              TopologyUtilities.CHECKPOINT_STREAM_ID,              new
Values(latestTimestamp));
        verifyNoMoreInteractions(collector);
    }

    @Test
    public void shouldCheckpointState() {
        // given
        BaseRichBolt                  component                  =                  mock(BaseRichBolt.class,
withSettings().extraInterfaces(IHasProcessingState.class));
        ProcessingState ps = new ProcessingState(new DateTime(), "SOME_SERIALIZED_STATE");
        when(((IHasProcessingState)component).getProcessingState()).thenReturn(ps);
        Jedis jedis = mock(Jedis.class);
        RecoverableBoltDecorator bolt = new RecoverableBoltDecorator(component, 1, jedis);
        Map conf = mock(Map.class);
        TopologyContext context = mock(TopologyContext.class);
        when(context.getThisTaskId()).thenReturn(1);
        OutputCollector collector = mock(OutputCollector.class);
        bolt.prepare(conf, context, collector);

        // when
        bolt.execute(MockTupleHelpers.mockTickTuple());

        // then
        verify((IHasProcessingState)component, times(1)).getProcessingState();
        verify(jedis, times(1)).set(eq("1"), eq(StringSerialzer.serialize(ps)));
    }

    @SuppressWarnings("unchecked")
    @Test
    public void shouldRequestBufferStateOnPrepare() {
        // given
        BaseRichBolt component = mock(BaseRichBolt.class);
        RecoverableBoltDecorator bolt = new RecoverableBoltDecorator(component, 1);
        Map conf = mock(Map.class);
        TopologyContext context = mock(TopologyContext.class);
        Map<GlobalStreamId, Grouping> grouping = mock(Map.class);
        when(grouping.keySet())
                .thenReturn(new      HashSet<GlobalStreamId>(Arrays.asList(new      GlobalStreamId("split",
Utils.DEFAULT_STREAM_ID))));
        when(context.getThisSources())
                .thenReturn(grouping);
        when(context.getComponentTasks("split")).thenReturn(new
ArrayList<Integer>(Arrays.asList(1)));
        OutputCollector collector = mock(OutputCollector.class);

        // when
        bolt.prepare(conf, context, collector);

        // then
        verify(collector).emitDirect(1, TopologyUtilities.REQUEST_BUFFER_STATE_STREAM_ID, new
Values(TopologyUtilities.REQUEST_BUFFER_STATE_PAYLOAD));
    }

    @Test
    public void shouldRequestProcessingStateOnPrepare() {
        // given
        BaseRichBolt component = mock(BaseRichBolt.class,
withSettings().extraInterfaces(IHasProcessingState.class));
        Jedis jedis = mock(Jedis.class);
        ProcessingState ps = new ProcessingState(new DateTime(), "SOME_SERIALIZED_STATE");
        when(jedis.get("1")).thenReturn(StringSerialzer.serialize(ps));
        RecoverableBoltDecorator bolt = new RecoverableBoltDecorator(component, 1, jedis);
        Map conf = mock(Map.class);
        TopologyContext context = mock(TopologyContext.class);
        when(context.getThisTaskId()).thenReturn(1);
        OutputCollector collector = mock(OutputCollector.class);

        // when
        bolt.prepare(conf, context, collector);

        // then
        verify((IHasProcessingState)component, times(1)).setProcessingState(refEq(ps));
    }

    @Test
    public void shouldDiscardReplayedTuplesPresentInRecoveredProcessingState() {
        // given
        BaseRichBolt component = mock(BaseRichBolt.class,
withSettings().extraInterfaces(IHasProcessingState.class));
        Jedis jedis = mock(Jedis.class);
        ProcessingState ps = new ProcessingState(new DateTime(), "SOME_SERIALIZED_STATE");
        when(jedis.get("1")).thenReturn(StringSerialzer.serialize(ps));

        RecoverableBoltDecorator bolt = new RecoverableBoltDecorator(component, 1, jedis);
        Map conf = mock(Map.class);
        TopologyContext context = mock(TopologyContext.class);
        when(context.getThisTaskId()).thenReturn(1);
        OutputCollector collector = mock(OutputCollector.class);

        bolt.prepare(conf, context, collector);
        // when
        bolt.execute(mockRecoveryTuple(new DateTime().minusSeconds(5), "should be discarded"));
        Tuple validRecoveryTuple = mockRecoveryTuple(new DateTime(), "should not be discarded");
        bolt.execute(validRecoveryTuple);

        // then
        verify(component, times(1)).execute(any(Tuple.class));
    }

    @Test
    public void shouldNotDiscardRecoveryTuplesIfNoProcessingStateToRestore() {
        // given
        BaseRichBolt            component            =            mock(BaseRichBolt.class,
withSettings().extraInterfaces(IHasProcessingState.class));
        Jedis jedis = mock(Jedis.class);
        when(jedis.get("1")).thenReturn(null);

        RecoverableBoltDecorator bolt = new RecoverableBoltDecorator(component, 1, jedis);
        Map conf = mock(Map.class);
        TopologyContext context = mock(TopologyContext.class);
        when(context.getThisTaskId()).thenReturn(1);
        OutputCollector collector = mock(OutputCollector.class);

        bolt.prepare(conf, context, collector);

        // when
        bolt.execute(mockRecoveryTuple(new DateTime().minusSeconds(5), "should not be discarded"));
        Tuple validRecoveryTuple = mockRecoveryTuple(new DateTime(), "should not be discarded");
        bolt.execute(validRecoveryTuple);

        // then
        verify(component, times(2)).execute(any(Tuple.class));
    }

    @Test
    public void shouldAckRecoveryTuplesAfterReplay() {
        // given
        BaseRichBolt            component            =            mock(BaseRichBolt.class,
withSettings().extraInterfaces(IHasProcessingState.class));
        Jedis jedis = mock(Jedis.class);

        RecoverableBoltDecorator bolt = new RecoverableBoltDecorator(component, 1, jedis);
        Map conf = mock(Map.class);
        TopologyContext context = mock(TopologyContext.class);
        when(context.getThisTaskId()).thenReturn(1);
        OutputCollector collector = mock(OutputCollector.class);

        bolt.prepare(conf, context, collector);

        DateTime timestamp = new DateTime();
        bolt.execute(mockRecoveryTuple(timestamp, "some payload"));

        // when
        bolt.execute(MockTupleHelpers.mockTickTuple());

        // then
        verify(collector).emitDirect(1,              TopologyUtilities.CHECKPOINT_STREAM_ID,              new
Values(timestamp));
        verifyNoMoreInteractions(collector);
    }
}
```

14.2.22.TestCollector.java

```java
package ts;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import org.testng.annotations.Test;
import ts.experiments.wordcount.storm.StormSentenceSpout;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import static org.fest.assertions.api.Assertions.assertThat;
import static org.mockito.Mockito.mock;


class TestCollector extends SpoutOutputCollector {
  public List<Object> Ids = new ArrayList<Object>();

  public TestCollector() {
    super(mock(SpoutOutputCollector.class));
  }

  @Override
  public List<Integer> emit(List<Object> tuple, Object messageId) {
    Ids.add(messageId);
    System.out.println(messageId);
    System.out.println(tuple.get(0));
    return null;
  }
}

public class SentenceSpoutTest {
  @Test
  public void shouldNotDuplicateIds() {
    StormSentenceSpout spout1 = new StormSentenceSpout();
    TestCollector collector1 = new TestCollector();
    spout1.open(mock(Map.class), mock(TopologyContext.class), collector1);
    StormSentenceSpout spout2 = new StormSentenceSpout();
    TestCollector collector2 = new TestCollector();
    spout2.open(mock(Map.class), mock(TopologyContext.class), collector2);

    for(int i = 0; i < 1; i++) {
      spout1.nextTuple();
      spout2.nextTuple();
    }

    assertThat((Long)collector1.Ids.get(0) == 0);
    assertThat((Long)collector2.Ids.get(0) == 1);
  }
  @Test
  public void shouldAckTuples() {
    StormSentenceSpout spout1 = new StormSentenceSpout();
    TestCollector collector = new TestCollector();
    spout1.open(mock(Map.class), mock(TopologyContext.class), collector);

    spout1.nextTuple();
    spout1.nextTuple();
    spout1.ack(collector.Ids.get(0));

    System.out.println();
  }

  @Test
  public void shouldFailTuples() {
    StormSentenceSpout spout1 = new StormSentenceSpout();
    TestCollector collector = new TestCollector();
    spout1.open(mock(Map.class), mock(TopologyContext.class), collector);

    spout1.nextTuple();
    spout1.nextTuple();

    spout1.fail(collector.Ids.get(0));

    spout1.nextTuple();

    System.out.println();
  }

  @Test
  public void shouldSendFailSignal() {
    StormSentenceSpout spout1 = new StormSentenceSpout();
    TestCollector collector = new TestCollector();
    spout1.open(mock(Map.class), mock(TopologyContext.class), collector);

    spout1.nextTuple();

    // Debug pause and wait for 20 secs

    // Set "ts-start-test" key in redis

    spout1.nextTuple();

    // Check to see that the signal tuple was sent
    System.out.println();
  }
}
```

14.2.23.TupleBufferTest.java

```
package ts;

import backtype.storm.tuple.Values;
import org.joda.time.DateTime;
import org.testng.annotations.Test;

import java.util.Arrays;
import java.util.List;

import static org.fest.assertions.api.Assertions.assertThat;


public class TupleBufferTest {

  @Test
  public void shouldBufferMultipleTuplesForSameTimestamp() {
    TupleBuffer buffer = new TupleBuffer(Arrays.asList(1));
    DateTime stamp1 = new DateTime();

    buffer.add(1, stamp1, new Values("first"));
    buffer.add(1, stamp1, new Values("second"));

    List<Values> tuples = buffer.getTuplesForTask(1);
    assertThat(tuples.size()).isEqualTo(2);
  }

  @Test
  public void shouldTrimTuples() {
    TupleBuffer buffer = new TupleBuffer(Arrays.asList(1));
    DateTime stamp1 = new DateTime();

    buffer.add(1, stamp1, new Values("first"));
    buffer.add(1, stamp1.plusSeconds(20), new Values("second"));

    buffer.trim(1, stamp1.plusSeconds(10));

    List<Values> tuples = buffer.getTuplesForTask(1);
    assertThat(tuples.size()).isEqualTo(1);
  }
}
```

14.3. Appendix C - Test results

Please refer to the CD submitted with this project for copies of the experiment logs and the results used in this report. The files on the CD are split into phases 1 and 2, corresponding to sections 9.2 and 9.3 of this report.