Towards Autonomic Fault Recovery in System-S

Gabriela Jacques-Silva*
Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL, USA

Jim Challenger, Lou Degenaro, James Giles, Rohit Wagle
IBM Research
T. J. Watson Research Center
Hawthorne, NY, USA

Abstract

System-S is a stream processing infrastructure which enables program fragments to be distributed and connected to form complex applications. There may be potentially tens of thousands of interdependent and heterogeneous program fragments running across thousands of nodes. While the scale and interconnection imply the need for automation to manage the program fragments, the need is intensified because the applications operate on live streaming data and thus need to be highly available. System-S has been designed with components that autonomically manage the program fragments, but the system components themselves are also susceptible to failures which can jeopardize the system and its applications.

The work we present addresses the self healing nature of these management components in System-S. In particular, we show how one key component of System-S, the job management orchestrator, can be abruptly terminated and then recover without interrupting any of the running program fragments by reconciling with other autonomous system components. We also describe techniques that we have developed to validate that the system is able to autonomically respond to a wide variety of error conditions including the abrupt termination and recovery of key system components. Finally, we show the performance of the job management orchestrator recovery for a variety of workloads.

1. Introduction

Stream processing has recently gained interest as a new way to analyze streaming data such as audio, video, chat, voice-over-IP, and email for applications ranging from monitoring customer service satisfaction to fraud detection in the financial industry. Being able to analyze data as it streams rather than storing and using data mining techniques offers the promise of more timely analysis as well as allowing more data to be processed with fewer resources. In this paper, we describe some of the autonomic self-healing capabilities of a stream processing system we are prototyping called System-S [1, 14, 10].

In System-S, the observation has been made that streaming analysis lends itself well to componentization and distribution. In particular, large and complex streaming analysis applications can be broken up into small software building blocks that we call *process elements* (*PEs*). For example, a filtering *PE* may consume a first stream and produce a second filtered stream that is consumed by a correlation *PE* that correlates the filtered stream with a third stream of data. This composition of applications by connecting *PE* building blocks together is a powerful feature because it allows for rapid development and introduction of new types of analysis by creating new *PEs* that are able to produce and consume streams. Even more importantly, this division of applications into building blocks connected by streams provides a natural way to distribute the computation task among a cluster of resources; *PEs* of different types can be placed on different nodes and the streaming data for *PEs* connected in the application can be communicated over the network. This distribution is particularly important for scaling in streaming applications because the algorithms involved can be compute intensive.

To better understand the complexity, we describe in more detail the architecture of the System-S runtime. System-S is designed to be a highly-scaled distributed system meant to support high volume stream processing. It is composed of a number of loosely-coupled runtime components and

*Sponsored by CAPES/Brazil

extensive tooling including an IDE to support composition of streaming jobs, a sophisticated planner, and visualization tools. The system runtime consists of a number of major components (see Figure 1):

- A framework upon which the job management, dispatching, and individual node controller components are built. We refer to the framework and the components built upon it as the job management component (JMN). The JMN itself is comprised of the central "orchestrator", the "resource manager" (RMN), and the "master node controller" (MNC) which will be described in more detail.
- an optimizer (OPT) which determines initial placement of processes and initial resource allocations, and continually adjusts both placement and allocations throughout the life-cycle of a job,
- a "stream processing core" (SPC) which manages the streaming communication system,
- a "dataflow graph manager" (DGM) which manages and tracks the graph of connections among the various streaming processes.

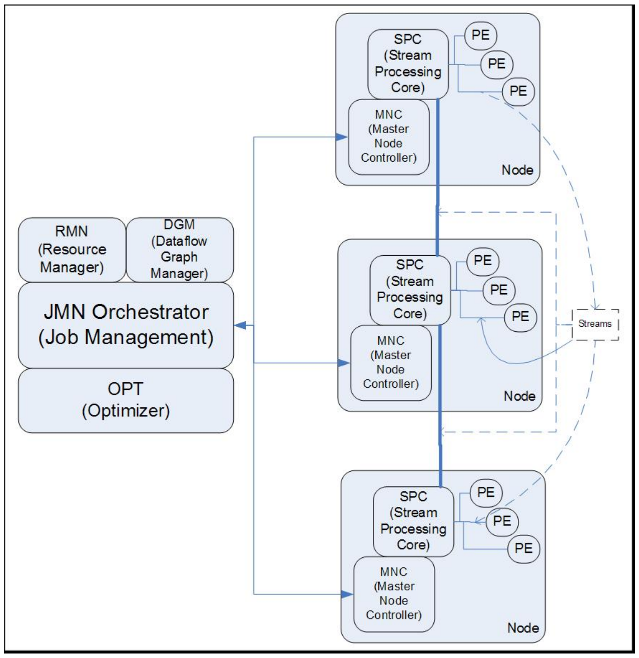

SPC (Stream Processing Core)
PE
PE
PE
MNC (Master Node Controller)
Node
RMN (Resource Manager)
DGM (Dataflow Graph Manager)
JMN Orchestrator (Job Management)
OPT (Optimizer)
SPC (Stream Processing Core)
PE
PE
PE
Streams
MNC (Master Node Controller)
Node
SPC (Stream Processing Core)
PE
PE
PE
MNC (Master Node Controller)
Node

Figure 1. System-S Architectural Overview

The purpose of the system is to support the scheduling, dispatch, and management of *streaming jobs*. A streaming job is defined to consist of one or more discrete *PEs* that use a highly optimized communication layer to exchange streaming data. *PEs* contain application logic and perform specialized functions such as filtering, annotating, segmenting, or joining of streams.

While streaming, componentization, and distribution all provide many advantages and capabilities for System-S, the features also introduce considerable management complexity. It is necessary for the system to distribute, start, stop, and recover individual *PEs* within the distributed system as well as the supporting system infrastructure. Because the applications require data to be streamed from *PE* to *PE*, the loss or failure of a single *PE* can cause an entire application to lose its ability to provide meaningful analysis until a repair is made. Further complicating the management task is the scale of the applications we have in mind that might incorporate tens of thousands of *PEs* running across thousands of heterogeneous nodes. To allow the streaming applications to continue to analyze live streaming data when there are failures of *PEs*, the infrastructure for such a system must have autonomic self-healing capabilities for both the application components and system infrastructure components; it is impractical for humans to keep the system running adequately without automation.

System-S had been designed with autonomics in mind. For example, as data changes occur such that there is suddenly less useful data on a stream, the system can shift resources to *PEs* processing other streams. As another example, if a node hosting a *PE* fails, the system can automatically allocate a new node, move the *PE* to the new node, and connect it to all of its producer and consumer streams. The work we present addresses the self healing nature of these management components in System-S. In particular, we show how the job management orchestrator can be abruptly terminated and then recover without interrupting any of the running program fragments by reconciling with other autonomous system components. We also describe techniques that we have developed to validate that the system is able to autonomically respond to a wide variety of error conditions including the abrupt termination and recovery of key system components. Finally, we show the performance of the job management orchestrator recovery for a variety of workloads.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 presents a technical overview of JMN. Section 4 describes the mechanism for self-healing of the JMN orchestrator such that the applications running in the system are not disrupted. Section 5 discusses a technique for validating the recovery mechanism under a variety of conditions. Section 6 describes the relative performance for recovery for a variety of system workloads. Section 7 provides future work and conclusions.

IEEE COMPUTER SOCIETY

2. Related Work

The JMN is being developed as part of System-S, an extreme-scale, streaming, data mining system described in Section 1. It is the control and management part of the distributed runtime, and must cooperatively interact with other parts of System-S. In particular, the JMN must interact with the stream processing core (SPC), dataflow graph manager (DGM), optimizer (OPT), resource manager (RMN), and supporting services such as security, an IDE, and cross-site management services.

Our work is also related to systems providing distribution management, autonomic management, and validation.

Distribution Management

The JMN is related to several existing batch schedulers [11] including Moab/Maui, LoadLeveler whose ancestor is Condor [17], Load Sharing Facility, Portable Batch System, Sun Grid Engine, and OSCAR. The high-level goal of these systems is to efficiently distribute and manage work across resources. While some of these systems are able to scale to large number of nodes and it is conceivable that they could scale to thousands of nodes, the nature of the System-S workload is fundamentally different than the workloads considered by these systems and it was not feasible to adapt them. Thus the JMN infrastructure was developed from scratch to support System-S requirements while leveraging state of the art technologies.

Autonomic Management

Because System-S is expected to scale to thousands of nodes and because of the interconnected nature of its applications, it must embody autonomic capabilities [15] to make management of the system feasible. For example, hardware failures are inevitable and the system must be able to adapt by reconstructing applications on other nodes to keep the analysis running. A related "Laundromat Model" [12] scheme for automatic management and process migration works by making the unit of control a virtual machine. This technique allows for migration of processes, but does not consider the relationships between them. Another related management infrastructure for stream processing [8] uses resource constraints and utility functions of business value to optimize application resource utilization. The Borealis system [2] lets users trade off between availability and consistency in a stream processing system by setting a simple threshold. Investigation of software failure and recovery in an autonomically managed environment has also been undertaken in the context of operating systems [4] and intrusion detection and recovery [16].

Validation

Key to any successful autonomic system is the ability to be reasonably certain that autonomic responses are appropriate. In many cases, it is difficult to formally verify appropriate responses because of the complexity of the algorithms involved in making autonomic decisions. Fault injection techniques have been developed to raise the confidence level in distributed systems.

One class of fault injection uses the message communication system to insert faults. Examples of such systems are ORCHESTRA [9] and FIONA (Fault Injector Oriented to Network Applications) [13]. The former is a local fault injector that inserts a protocol layer which filters messages between components in a distributed system. Every node in the system runs an independent instance of the tool. The latter is a distributed tool that alters the flow of UDP (User Datagram Protocol) messages in Java programs, and takes advantage of the underlying communication layer (OS layer) to coordinate an experiment to test a set of nodes. Both tools lack a broad fault model and also the ability to define precise triggers based on application state.

Another class of fault injection uses the operating infrastructure of a node. For example, NFTAPE (Network Fault Tolerance and Performance Evaluator) [18] presents a generic way to inject faults by writing a test script which can be used to inject faults on a remote node. Its design facilitates the injection of faults externally to the application (for example, through the operating system), making it more difficult to inject faults based on the application state. With NFTAPE, it is possible to use the processor breakpoint registers to define addresses at which faults will be injected. Loki [7] allows fault injection in multiple nodes based on a partial view of the application global state. The drawback of this approach is that the application has to be explicitly instrumented with state notifications and fault injection code. Also, a state machine must be defined to describe the distributed system and the global state in which the fault will be injected. Such tasks get more complicated when the system runs in a heterogeneous environment, where there is no guarantee concerning the language in which the applications are implemented and which state each of these pieces will be at each time interval. Multi-threaded applications where each thread has its own state may also cause problems when defining a state for a single process.

Building upon the related work cited above, the problem we address is self-recovery of the JMN orchestrator and the validation of the recovery technique.

3. Job Management Overview

This paper focuses on fault recovery of the job management component. The job management (JMN) framework is built around a highly modular "pluggable" design, similar in spirit to the Eclipse architecture [5]. During bootstrap the JMN framework initializes basic logging, network, and configuration services. The plugin manager is then invoked to dynamically load the pluggable functions which allow the JMN process to do useful work. Pluggable components provide all of the key features of JMN such as the external API, network protocols, persistence, system interfaces (e.g. to OPT, SPC, DGM, and so on), time services, security, and resource management.

Different pluggable functions are loaded into the framework to provide the following JMN processes:

- The central orchestrator, which provides traditional job management functions, fielding job submission and control, orchestration job life-cycles, persistence of job related information into the JMN database, and providing interfaces to most of the rest of the system. There is generally a single orchestrator in the system.

- The master node controller (MNC). The MNC runs on every node on which *PE*s are to be started. The MNC receives, executes, and manages start, stop, and various administrative orders from the orchestrator and provides the primary interfaces to the stream processing core (SPC).

- The resource manager, or RMN. The resource manager provides resource manager functions such as resource discovery, monitoring, and management.

The primary control mechanism of the job manager's orchestrator is the Finite State Machine (FSM) engine. The FSM engine itself does not define any automata. Rather, it provides a framework upon which automata may be defined and managed. It is possible to define automata consistent with Moore, Mealy, and hybrid FSM models.

Specific FSM implementations are defined externally via an XML document. The FSM Engine reads the document and constructs a specific instance of FSM from it. It is possible to construct and operate multiple, disparate FSM instances within the same process, and it is possible to assign different FSM instances to different jobs in the system if it is determined their life-cycles are different (for example, the life-cycle of a streaming job may be different from a non-streaming job).

Implementation of states and actions are done via the system's plug-in mechanism. An FSM instance consists of a collection of states. Figure 2 shows a portion of a FSM definition which specifies a single state called *optimizing*. A state consists of a collection of *transitions*. A *transition* consists of a number of *methods*. When a transition is triggered, each of the methods associated with it is executed in turn. When the last method returns, the state is advanced to the next state specified by the transition. It is possible to associate no methods with a transition, in which case state advances with no associated action. Each method specifies an *object*, which may be a plug-in, and a method on that object to execute. The methods must implement a specific interface which provides context for the execution of the method. Note also the ability to add arbitrary annotations to a state. These annotations may be queried by specific automata as needed. In the example we show the *updatedb* property which our FSM uses to determine whether the act of transitioning to this state should be persisted in the checkpoint database.

```
<state id="optimizing" number="5">
   <property name="updatedb" value="false"/>

   <transition id="optim" nextState="dgm-inst">
      <method object="fsmimpl" name="dgmInst" />
   </transition>

   <transition id="cancel" nextState="cleanup">
      <method object="fsmimpl" name="dgmUnmap" />
      <method object="fsmimpl" name="qCancel" />
   </transition>

   <transition id="error" nextState="cleanup">
      <method object="fsmimpl" name="dgmUnmap" />
      <method object="fsmimpl" name="qToCancel" />
   </transition>

   <transition id="recover" nextState="dgm-inst">
   </transition>
</state>
```

Figure 2. Subset of Finite State Machine Definition

Because the FSM definition is externalized, it is extremely easy to alter FSM behavior without code changes. This mechanism makes it possible to easily build the State-based Fault Injection system described in Section 5 that can be used to validate that the system is able to self-heal under a variety of error conditions.

4. Job Manager Orchestrator Recovery

As mentioned in Sections 1 and 3, the orchestrator is a central component of the system–all job submission and management flows through it. Hence it is also a single point of failure: a crash makes the system unavailable for submission of new work and management of existing work. However, since the system is distributed, failure of the orchestrator does not imply total system failure. If the system can self-heal by starting a new orchestrator and synchronizing its state with the nodes, it is possible for the rest of the system to be left unperturbed with no existing work lost. To accomplish this, we must implement a recovery technique which brings the orchestrator to a state consistent with all the components of the system. Such recovery techniques must also consider all state transitions of the distributed components which occurred during the outage, including their possible failure.

This section presents the fault model considered, the check-pointing techniques used and how the job manager orchestrator reconciles with the other components in the system.

4.1 Fault Model

The fault model considered includes failure of the orchestrator due to a physical node failure, crash of its process, or link failure (the node running orchestrator loses its network connection). While the recovery techniques could also be used for planned outages or migrations of the orchestrator, planned outages and migrations are more efficient using a controlled process not discussed in this paper.

Overall, our recovery approach is most closely aligned with the mixed-level checkpointing (MLC) [6] technique. Each individual *PE* is responsible for application level checkpointing (ALC) of its own state, while the JMN is responsible for system-level checkpointing (SLC) of each individual job's state (e.g., the collective state of a group of related *PEs*). We limit our discussion below to SLC and recovery.

4.2 State Check-pointing

As stated in Section 3, the job management architecture is a pure plug-in-based architecture with all useful functions implemented as plug-ins. Most of the plug-ins are stateless and thus do not need any recovery action. In fact, the only state that has to be maintained between JMN executions is the state of the jobs running in the system.

When a fault occurs, a job may be in any of the states defined by the externalized FSM description. Further events that trigger job state transitions could also occur at any time. A job state is dependent not only upon the state of the orchestrator itself, but also upon all the nodes that have *PEs* executing for that job. To save the state of each job, we use persistent objects, propagating the job state and its identification to a database. The state of each component of the job is inserted in the database using the Hibernate service [3].

While job persistence could be triggered every time there is a job state transition to maintain the most up to date state, persistence of states is performed selectively via the annotation mechanism described in Section 3. One reason for selective updates is the negative performance impact if we persist the updates generated by all the jobs in the system and all their distributed *PEs*. The second reason is that individual job state is dependent on the distributed state of its individual *PE*s. State change in an individual *PE* may not require a job state update, so job state is (re)persisted only when the collected state of its *PE*s indicates such an update. Hence, the state of the database when failure occurs may not reflect the actual state of the distributed system. In addition, individual *PE* states may change during the failure and recovery. Therefore, it is necessary to perform a reconciliation step after JMN orchestrator failure using the database state as a starting point to recover each job. Different reconciliation procedures are used for each job (for example, undo/redo or state forwarding), depending on the state of the job at the time of the failure.

Figure 3 shows a simplified state diagram of a job. When a job is submitted to the system, its data are persisted to the database and the job state transitions to the *parsing* state where the job description is interpreted. Until the job state arrives at the *mapping* state, only the orchestrator is aware of the job. In the *mapping* state, the orchestrator requests the Dataflow Graph Manager (DGM) to map the potential streaming connections among *PEs*. The job state then transitions to the *optimizing* state, where node placement decisions are made, and then to the *instantiating* state, where DGM is contacted once more to inform it of the node assignments for the *PEs*. In the *dispatching* state, the orchestrator contacts each of the nodes assigned to the job to begin execution of the *PE*s. From this point, the job state is dependent on the collective distributed state of the *PE*s. When all *PEs* have started, the job state transitions to the *running* state. After the *PEs* are finished, the job completes. By analyzing the impact of a fault in each state, it was determined that the states *submitted, mapping, dispatching, canceling* and *running* should be persisted to the database. The states were selected based on which states could lead JMN to an incosistency in case of a failure. If data can be recovered from other components of the system, we only save the first time the state changes on JMN. For example, if the job was in *running* state and receives a notification of a *PE* exit, the job will still be in *running* state waiting for other *PEs* to finish, and, therefore, will not persist such transition again. States that do not interfere with other components and can be repeated internally to JMN are not saved.

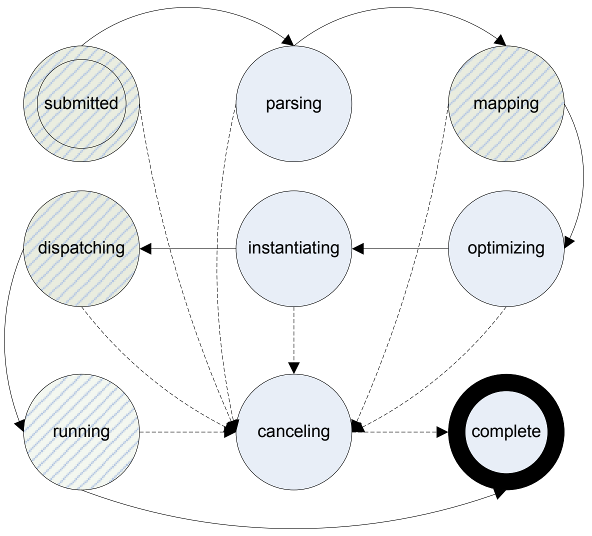


Figure 3. Job state machine

It is important to notice that if a job misses a *PE* state transition, the FSM may stall. The MNCs normally communicate with the orchestrator through asynchronous status update reports. If the orchestrator fails, the MNCs detect message failure and keep these messages on a FIFO list for timed retransmission. During recovery, the orchestrator's communication mechanisms are fully functional, but the status listener returns error status to any such messages that arrive, which insures they remained enqueued in the MNC's retry lists. Once the orchestrator is able to process the messages it removes the block from the status listener and after at most one retry cycle all nodes will have been able to report all pending status, (in the correct order).

4.3 Recovery Plug-in

Consistent with the JMN architecture, the recovery module is implemented as a plug-in which is loaded when JMN is restarted after a crash/shutdown. The overall recovery process is:

1 query database for uncompleted jobs,

2 synchronize valid job state with DGM state if needed,

3 perform recovery action based on job state,

4 schedule a thread to finalize the recovery process (see Section 4.4).

The reconciliation process takes place in phases 2 and 3, when the orchestrator contacts other processes to reach a consistent state.

As mentioned in the previous section, the database is used as the starting point for orchestrator recovery. The management representation of all of the uncompleted jobs in the system is restored from the checkpoint and their object instances in the orchestrator are recreated. We enumerate the steps for recovering depending on the persisted states of *submitted, mapping, dispatching, canceling* and *running*.

Recovering when the persisted state is submitted

The first possible recovered persisted state is *submitted*. If the checkpoint indicates the job is in such a state, it means that the job could have already advanced to the parsing state before the failure as seen in Figure 3 (the canceling state and mapping state are not possible because they would have been persisted). Since the parsing state and submitted state do not depend on any distributed component, the recovery process can safely reset the job state to *submitted* so that the submission can be restarted.

Recovering when the persisted state is mapping

The second job state that can be recovered from the database is *mapping*. From the *mapping* state, the only possible un-persisted transitions before a failure are the optimizing and instantiating states (canceling and dispatching would have been persisted). Transition to the mapping state triggers an interaction with the DGM that advances the DGM's state. Completion of that interaction with DGM triggers a transition to the optimizing state which causes and interaction with OPT. During recovery, if a query of the DGM reveals that it does not have state for a job, then the job state can safely be returned to mapping to attempt the mapping operation with the DGM again. If instead the DGM has state for the job, then it is not possible to tell whether the mapping completed and the state was advanced to optimizing or instantiating before the failure or if the mapping did not complete. In either case, the available resources and PEs may have changed during the failure causing the map to no longer be valid. Since nothing has actually been dispatched and it is relatively cheap to start over in these cases, we clear the state from the DGM by calling unmap and then return the state of the job to mapping to trigger a new map with the DGM.

Recovering when the persisted state is dispatching

The next job state that can be recovered from the database is *dispatching*. If the job is in the dispatching state, it means that the orchestrator may or may not have made the requests to start each of the *PEs* on each of the nodes. The MNCs queue state transition messages while the orchestrator is unavailable. To recover, the orchestrator queries the MNCs and collects all of the messages that the MNC has queued as well as the last successful MNC state updates for each *PE*. If any *PE* of a job has a non-started status, the job resumes the dispatching steps. If all *PEs* are started, the job may move forward to the *running* state.

Recovering when the persisted state is running

If the last persisted state of a job was the *running* state, some or all of the *PEs* may have completed during the failure or recovery process. To recover in this case, the orchestrator has to contact the nodes to get their latest information about the *PEs* in the job. After the orchestrator gets all the responses, it can update the state of the job accordingly (keep it in *running* state if there are some *PEs* running or transition to *complete* state). If a *PE* is not known by its recorded MNC, it means that the *PE* has completed or the MNC has also failed during the orchestrator failure. In that case, the corresponding *PE* is marked as completed, and the job state advances depending on the state of all its *PEs*.

Recovering when the persisted state is canceling

If the persisted state of the job is canceling, some or all of the *PEs* may have completed during the failure as in the persisted running state. In that case, the nodes have to be queried to determine the current state of the constituent *PEs* so the job can be recovered as completed or held in the canceling state. As with the running state, if a *PE* is not known by its recorded MNC, then the *PE* has completed or the MNC has failed and restated. In either case, the *PE* is treated as complete and the job state advances accordingly.

Figure 4 shows the overall view of the recovery reconciliation process. During phase 1, the orchestrator queries the checkpoint database. After that, it reconciles with the DGM (phase 2) and recovers the jobs that were in *submitted* and *mapping* state. After these two phases, the orchestrator has information about all jobs in the system, but it may still have some inconsistencies. The figure shows Job 1 with two *PEs* started, and Job 2 with one *PE* started and another one not started. In phase 3, the orchestrator queries the MNCs about the last state of the constituent *PEs* and updates the job's overall state. After phase 3 the *PE* states are updated, one being completed and the other completed with exception. After finishing this phase, the orchestrator is consistent with all the components of the system and can start to operate normally (e.g., receiving new requests).

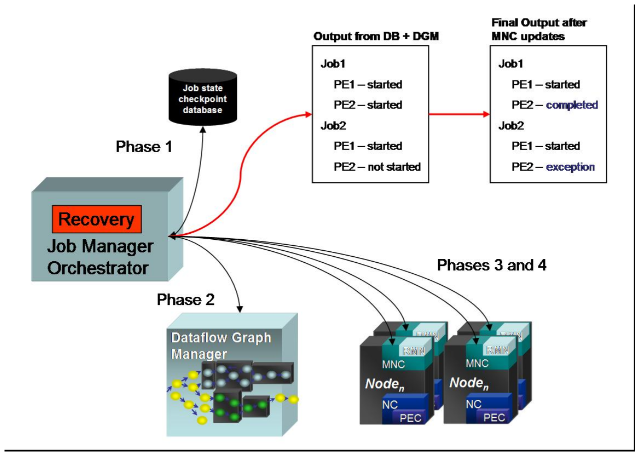


Figure 4. Recovery reconciliation process

4.4 Handling multiple failures

The JMN orchestrator depends on four remote entities to recover successfully: *PEs*, MNCs, a database manager system (DBMS) and the DGM. While the orchestrator is not running, any of those components may also have failed.

When a *PE* fails, the MNC detects it and report its status as "exception". This is handled via a state transition. The orchestrator may take further actions, such as moving the *PE* to another node, determined by system policy.

A node can fail in two situations of interest: before orchestrator failure/recovery and during orchestrator recovery. If failure occurs before the orchestrator begins its recovery it will be detected when communication to the node fails. If the node fails during the recovery process, the orchestrator may wait indefinitely for the response to the query of the *PEs* statuses. To successfully recover under this situation, the orchestrator schedules a thread that times out if a node is unresponsive (phase 4 in Figure 4). If the node is determined to have failed because of a timeout, the orchestrator treats this as a *PE* failure and applies system policy to the job to determine the recovery strategy (e.g. cancel the job, restart the *PE* elsewhere, etc.).

The orchestrator relies on the DBMS for checkpoint availability and integrity. If the DBMS fails, the orchestrator relies on the fault tolerance mechanisms of the DBMS itself. Similarly, the DGM has its own fault tolerance mechanism. Note, however, that the orchestrator can not operate without the DBMS or DGM, so any orchestrator recovery strategy may require waiting for recovery of these processes before its own recovery can begin.

5. Recovery Validation

As a means to validate the orchestrator recovery mechanism to obtain the high availability requirements, we need techniques to evaluate the recovery time and assure that the mechanism developed can diagnose and react to faults correctly. The ability of the system to survive under various abnormal behaviors of all the participating components distributed across a network of nodes is a challenge. In this section we present the fault injection tool developed to help validate the JMN orchestrator recovery approach. The technique can also be used to validate other recovery mechanisms within System-S, as well as any distributed system controlled by an externalized finite state machine.

As mentioned in Section 3, JMN performs scheduling and dispatching of jobs based on an externalized finite state machine (FSM), which can be expressed using a mark-up language, such as XML (Extensible Markup Language). Our fault injection tool *FSM Fault Injector* (FSMFI) leverages the fact that the FSM in the JMN framework is externalized by augmenting the FSM definition with the faults. This approach permits the test engineer to inject errors based on the system state and also facilitates injection of errors in other nodes of the distributed system. The advantage of such a tool is that it can precisely exercise a distributed system under a variety of faulty conditions. Since each fault injection occurs explicitly in system states, testing can be more targeted with greater confidence in the test coverage.

To ensure that the operational FSM does not have errors introduced by fault injection, the test engineer creates a *fault injection test* (FIT) document which defines a fault injection campaign in a standardized format (e.g., XML document) and specifies the states and the transitions in which the fault injection will take place. The faulty behavior can be chosen from a fault injection library or defined by the tester. After the definition of the FIT document, the FSMFI tool creates an augmented FSM description that is modified with fault injection annotations. The modified FSM is loaded and the FSM Engine calls the fault injection methods when appropriate. Using this technique the tester can easily add, remove and change faulty behavior without making changes

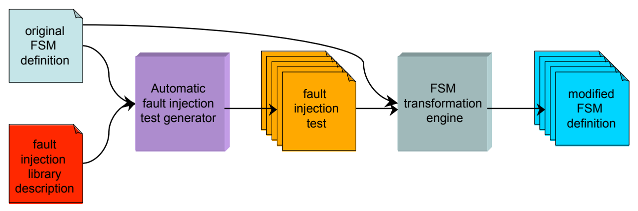
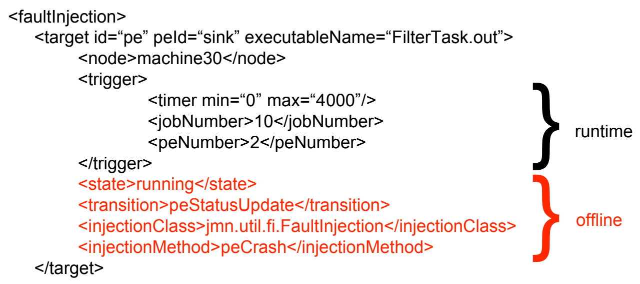

original FSM definition
fault injection library description
Automatic fault injection test generator
fault injection test
FSM transformation engine
modified FSM definition

Figure 5. Automatic fault injection test generation



```
<faultInjection>
    <target id="pe" peId="sink" executableName="FilterTask.out">
        <node>machine30</node>
        <trigger>
            <timer min="0" max="4000"/>
            <jobNumber>10</jobNumber>
            <peNumber>2</peNumber>
        </trigger>
        <state>running</state>
        <transition>peStatusUpdate</transition>
        <injectionClass>jmn.util.fi.FaultInjection</injectionClass>
        <injectionMethod>peCrash</injectionMethod>
    </target>
```

runtime
offline

to the application. Figure 5 demonstrates how the original FSM description of the distributed application and the fault injection library description are used as input for a automatic fault injection test generator. The output is a collection of FIT documents which describe the various fault injection tests. These FIT documents are used as input, along with the FSM description, to a FSM transformation engine, which generates a modified FSM definition.

The FIT language for specifying the fault injection test is comprised of a description of the faults to be injected as well as information that implies a modification directly to the FSM and information related to the configuration of the fault injection library (e.g., time trigger). It is described in a standardized format according to a fault injection schema. The implementation of fault injection methods, which emulate the faulty behavior desired by the tester, may be through the use of pre-implemented methods from a fault injection library or plug-in. These fault-providing methods may accept runtime configuration options described in the fault injection test XML document. In our prototype, the library is implemented as a fault injection plug-in.

Figure 6 shows an example of a fault injection campaign description. It shows both *runtime* options, used by the fault injection library, and *offline* options used by the FSM transformation engine. In this example, the faulty behavior is implemented by the method `peCrash` in the `jmn.util.fi.FaultInjection` class. The target of the fault injection is a subtask (`FilterTask.out`) executing in `machine30`. The trigger is based on the state of a job and optionally also by a timer. Here the fault is triggered when the job is in the `running` state and it makes a `peStatusUpdate` transition. Since there is also a timer configuration, the fault is injected at a random time between 0 and 4000 milliseconds after the transition is taken.

Figure 7 shows how the FSM description is modified by a configuration as shown in Figure 6. Originally, when a job is in the `running` state and makes a `peStatusUpdate` transition, it executes in sequence `method1`, `method2` and `method3`. After processing the original FSM with the fault injection description, the new FSM is generated with the fault injection method inserted in the state transition. During the scheduler execution, when such a transition is taken, it will invoke the fault injection method and, in this

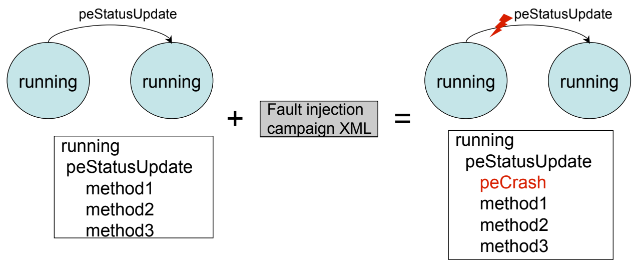


Figure 7. FSM description modified for a fault injection experiment

case, inject a *PE* crash fault, according to the configuration passed to the fault injection plug-in.

Using the FSMFI mechanism, tests of the JMN orchestrator recovery mechanism can be automatically constructed from the original externalized FSM and the fault injection library. While the approach requires use of an externalized FSM, we believe that the benefits for validation such as granularity and repeatability are compelling. In the next section we show how recovery was validated for some states using the FSMFI technique.

6. Results and Experiments

This section describes qualitative experiments with the orchestrator recovery technique driven by different kinds of fault injection techniques. We also show the relative performance of the recovery technique versus restarting the applications if the orchestrator fails.

The experiments were conducted with System-S running on several Linux nodes. One node was running the JMN orchestrator, another one was dedicated to the DGM. All nodes were running MNCs and executing *PEs*. To reduce complexity, a random placement mechanism was used instead of the optimizer. The check-pointing database was running on the same node as the orchestrator and the DBMS used was HSQLDB (`http://hsqldb.org`).

For all experiments, the System-S infrastructure is first started on all nodes. After the orchestrator, DGM, and all MNCs are running, jobs are submitted to the system. For each of the experiments, under different conditions, a crash fault is injected into the orchestrator. After the orchestrator crashes, it is restarted in recovery mode. In the first experiment, the crash of the orchestrator occurs when the system is in steady state, while in the second experiment the crash occurs in the middle of a job dispatch.

6.1 Crash in a Steady State

This first experiment shows the recovery of the orchestrator while in a steady state. Steady state here means that no jobs are being submitted to the system during the time of crash and all the jobs already in the system are in running state, so no new transitions related to job start are being sent to the orchestrator. We submitted multiple jobs, each containing 31 *PEs*, out of which there was one source PE and 13 sink *PEs*. The placement of the PE's is chosen at random by the system after job submission. Another test with a set of 104 jobs containing a total of 737 *PEs* was also carried out. These jobs have some *PEs* that are shared among multiple jobs and also have stream connections between *PEs* of different jobs. This test set is used because it more closely represents real world systems (RWS).

The experiment begins by starting up the system and submitting a specific number of jobs. We then introduce a crash by causing the JMN orchestrator to abruptly stop when all the jobs are running in the system. The crash is introduced using the FSM based fault-injection technique described in the previous section . The *PEs* continue to run in spite of the orchestrator crash, but no new jobs can enter the system and the jobs that are running cannot be cancelled.

After the JMN orchestrator crashes, it is restarted in recovery mode. We observed that during the recovery it successfully contacts all MNCs. It also updated the state of all the jobs and the *PEs* according to the state saved and reported by each MNC. After the recovery is complete, new jobs can be submitted and further updates for existing jobs can be processed normally.

The above process of crashing and restarting JMN orchestrator is carried out multiple times. Time to recover in each case is noted. Another set of tests are carried out where we start up the system and submit jobs. We then cancel the jobs and re-submit them. The time it takes in each case from the moment each job is submitted till the last job enters running state is noted.

6.2 Crash while Dispatching a Job

The second experiment shows that the orchestrator is able to recover when a job is in the dispatching state. In this experiment 3 jobs were submitted, where one had 9 *PEs* and the other two had 32 *PEs*.

The state-based fault injection technique was used to modify the FSM dispatching state so that the FSM engine invokes the `crashJmn` method while it is dispatching the third job after half of its *PEs* are dispatched.

During the recovery, the orchestrator detects that one of the jobs is still in dispatching state, so it tries to finish the dispatch. The orchestrator reports that the job successfully completes the dispatch and the orchestrator is ready to receive new job requests.

6.3 Experimental Results

We have described the experiments in which the recovery process is confirmed for faults when the system is in steady state and when the system is in the middle of dispatching a job. We have also described that recovering from failure autonomically with minimal interruption to applications is critical for stream processing systems. In this subsection we show the performance of recovery is also favorable compared to restarting the applications from scratch.

Test #	*# of nodes*	*# of PEs*	*secs. to start jobs*	*secs. to recover*
1	10	31	2.0	7.6
2	10	310	9.1	8.8
3	30	310	9.1	8.0
4	30	930	26.3	15.1
5	90	930	33.9	16.2
6(RWS)	90	737	27.0	15.4

Figure 8. Job Submission vs. Recover Times (Steady State Crash)

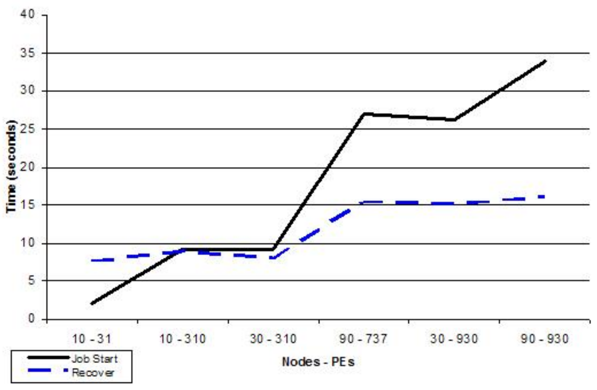


Figure 9. Comparison of Job Start versus Recovery Times (Steady State Crash)

Notes:

1. Seconds to startup jobs only includes time since the first job was submitted until all jobs were running.

2. Seconds to recover includes only orchestrator boot time plus time needed to recover all PEs as running.

3. For each node and *PE* number combination shown above, the experiment was conducted 10 times. Average times are shown.

The table and chart shows that for very small number of *PEs*, the recovery time is more than the start time. However as the number of *PEs* in the system goes up, the recovery time increases very slowly as compared to job startup times.

7. Conclusions and Future Work

This paper has described the self-healing capabilities of a key component of the System-S infrastructure and presented techniques for validating the recovery mechanism. One important contribution is the realization that persistence overhead could be reduced in System-S by persisting a subset of the state transitions in the lifecycle of a job and rolling back the work to an appropriate state, redoing a transition, or advancing to an appropriate state depending on the state recovered and environmental conditions. Also, the checkpointing scheme on JMN and MNCs do not require synchronization. Therefore, when the recovery takes place, there is no concern with reaching strong consistency between the distributed components. Another important contribution is the description of the externalized FSM which can make it easy to programmatically inject faults that are dependent on the state of an application. We have described two qualitative experiments that show the job manager orchestrator can recover under a variety of conditions. We have also explained why it is important to recover streaming applications without interrupting them and by experimental analysis have shown that recovery is in general faster than restarting the system and applications from scratch. Future work includes the conduction of comprehensive fault injection tests, evaluating the resilience of JMN to other fault models (e.g. bit flips) and fault propagation behavior.

References

[1] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 71, Washington, DC, USA, 2006.

[2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of ACM SIGMOD '05*, pages 13–24, New York, NY, USA, 2005.

[3] C. Bauer and G. King. *Hibernate in Action*. Manning Publications, New York, NY, 2005.

[4] A. Bohra, I. Neamtiu, and F. Sultan. Remote repair of operating system state using backdoors. In *Proc. of ICAC '04*, pages 256–263, Washington, DC, USA, 2004. IEEE Computer Society.

[5] A. Bolour. Notes on the eclipse plug-in architecture. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[6] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Recent advances in checkpoint/recovery systems. In *Workshop on NSF Next Generation Software*, 2006.

[7] R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders. A global-state-triggered fault injector for distributed system evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605, July 2004.

[8] B. F. Cooper and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proc. of ICAC '05*, pages 3–14, Washington, DC, USA, 2005.

[9] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A probing and fault injection environment for testing protocol implementations. In *Proc. of IPDS'96*, Urbana-Champaign, IL, 1996.

[10] F. Douglis, M. Branson, K. Hildrum, B. Rong, and F. Ye. Multi-site cooperative data stream analysis. *SIGOPS Oper. Syst. Rev.*, 40(3):31–37, 2006.

[11] Y. Etsion and D. Tsafrir. A short survey of commercial batch schedulers. Technical Report 13, May 2005.

[12] J. G. Hansen, E. Christiansen, and E. Jul. The laundromat model for autonomic cluster computing. In *Proc. of ICAC '06*, pages 114–123, June 2006.

[13] G. Jacques-Silva, R. J. Drebes, J. Gerchman, J. M. F. Trindade, T. S. Weber, and I. Jansch-Pôrto. A network-level distributed fault injector for experimental validation of dependable distributed systems. In *Proc. of COMPSAC '06*, pages 421–428, Chicago, IL, USA, 2006.

[14] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proc. of ACM SIGMOD '06*, pages 431–442, New York, NY, USA, 2006. ACM Press.

[15] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, January 2003.

[16] H.-H. S. Lee, G. Gu, and T. N. Mudge. An intrusion-tolerant and self-recoverable network service system using a security enhanced chip multiprocessor. In *Proc. of ICAC '05*, pages 263–273, Washington, DC, USA, 2005.

[17] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor–a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, 1988.

[18] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proc. of the IEEE IPDS 2000*, pages 91–100, Chicago, USA, March 2000.