Implementation and Evaluation of a Protocol for Recording Process Documentation in the Presence of Failures

Zheng Chen and Luc Moreau

School of Electronics and Computer Science
University of Southampton, Southampton, SO17 1BJ, UK
{zc05r,L.Moreau}@ecs.soton.ac.uk

Abstract. The *provenance* of a particular data item is the process that led to that piece of data. Previous work has enabled the creation of detailed representation of past executions for determining provenance, termed *process documentation*. However, current solutions to recording process documentation assume a failure free environment. Failures result in process documentation not being recorded, thereby causing the loss of evidence that a process occurred. We have designed F-PReP, a protocol to guarantee the recording of process documentation in the presence of failures. This paper discusses its implementation and evaluates its performance. The result reveals that it introduces acceptable overhead.

1 Introduction

The *provenance* of a data product refers to the process that led to that data product [6]. Previous work [6] has enabled a computer-based representation of a past process for determining provenance, i.e., *process documentation*. A dedicated repository, *provenance store*, is used to persistently maintain process documentation. For scalability reasons, process documentation may end up distributed in multiple stores, linked by pointers. Using the pointer chain, distributed process documentation can be retrieved from one store to another.

A generic recording protocol, PReP [6], has been developed to record process documentation in Grids. It has been used in many applications, e.g., aerospace engineering [8], fault tolerance in distributed systems [15], and biodiversity [14]. Grids are large-scale heterogeneous environments, where failures may happen. Failure rates as high as 30% have been reported [13]. In this context, reliable recording of process documentation can become very challenging, given that the documentation produced in a process can be of the order of terabytes [5].

PReP, however, does not specify a well-defined behavior to record process documentation in the presence of failures. For example, a provenance store may not be available and network connection may be broken. The consequences are that documentation fails to be recorded in provenance stores and the pointer chain is broken, separating distributed documentation into isolated islands in provenance stores. A scientific application, to be described in this paper, used

J. Freire, D. Koop, and L. Moreau (Eds.): IPAW 2008, LNCS 5272, pp. 92–105, 2008.

PReP to record process documentation in the presence of simulated failures. By analyzing the contents of provenance stores after the application completes, we find that the quality of documentation recorded using PReP is poor, as demonstrated in Fig. 1 and Fig. 2. In Fig. 1, as failure rate increases, a large proportion of process documentation fails to be recorded. Fig. 2 reveals the increase in the number of dangling links, i.e., pointers to other provenance stores that were supposed to record part of process documentation but did not, and in the number of isolated documentation islands.

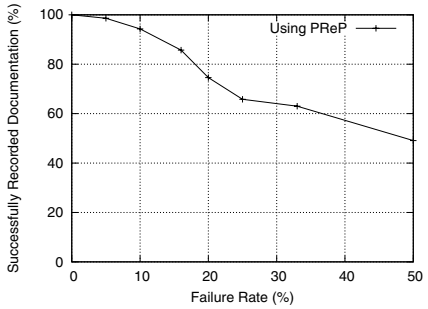

Fig. 1. Loss of documentation records in provenance stores

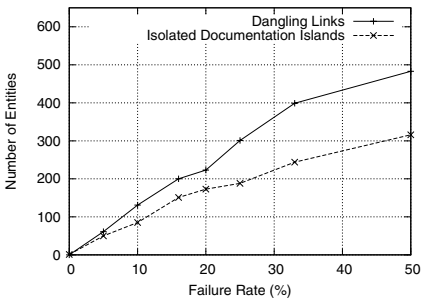

Fig. 2. Dangling links and isolated islands in provenance stores

Process documentation of poor quality cannot be utilized by applications. We now draw a parallel between the documentation of a process and a particular type of evidence in a legal setting, testimony. The absence of testimony from eyewitnesses to a crime scene makes it difficult for juries to make a judgment about whether to believe the claims provided by a suspect. Similarly, poor quality process documentation is not acceptable in the applications that rely on process documentation to verify the provenance of their data products, as key evidence that a process occurred may have been lost.

To guarantee the recording of *high quality*, i.e., complete and connected, process documentation in Grids where failures may occur, we have extended PReP and designed a recording protocol, F-PReP. F-PReP provides remedial actions and a novel component, Update Coordinator. It has been formalized as an abstract state machine and its correctness has been proved in [4].

The contribution of this paper is the extensive evaluation of this novel protocol. Our evaluation is conducted at several levels. First, we measure the throughput of the provenance store and update coordinator. We demonstrate that the update coordinator is not a performance bottleneck. Second, we benchmark the recording performance of F-PReP. The results show that its remedial actions introduce small overhead (below 10%). Third, we investigate the performance impact on the execution time of a scientific application. We find that PReP and F-PReP have similar impact on application execution when there is no failure. In tests with failures, the recording overhead of F-PReP varies depending on configurations. Lessons are learned on achieving good performance in the case

of failures. Our results also show that the problems in Fig. 1 and Fig. 2 do not exist when using F-PReP to record process documentation.

2 Protocol Outline

2.1 Terminology

A process is modeled as a set of interactions between actors [6]. Each *interaction* is concerned with one application message exchanged between two actors, i.e., the sender and the receiver. An actor documents an interaction by making *p-assertions*. A kind of p-assertion, *relationship p-assertion*, is used to capture the internal causal connections between interactions within the scope of an actor, i.e., the interaction where an output message is sent (*effect interaction*) and the interaction where an input message is received (*cause interaction*). There can be multiple cause interactions related to one relationship p-assertion.

For scalability reason, an actor can use various stores to record p-assertions about different interactions. A notion of link, i.e., a pointer to a provenance store, has been introduced to connect distributed documentation [6].

There are two types of links: *viewlink* and *causelink*. If the two actors in an interaction use two different stores, each actor records a *viewlink* that points to the provenance store where the opposite party recorded its p-assertions about that interaction. Therefore, both views of an interaction can be retrieved by navigating from one store to the other. The causelink is used in relationship p-assertion. If the p-assertions that represent a cause interaction are recorded in a different provenance stores, a *causelink* is embedded in the relationship p-assertion, indicating which provenance store the p-assertions representing the cause interaction are stored in. A relationship p-assertion is recorded in the context of its effect interaction, describing the causes that led to the effect.

2.2 Failure Assumptions

Provenance stores may crash, i.e., they halt and stop any further execution, and can be restarted from their latest consistent state[1]; messages to/from provenance stores can be lost, reordered but not duplicated in communication channels; an actor has several alternative provenance stores to use. We do not consider the failures of actors and the exchange of application messages, since they are application dependant and the application should provide its own fault tolerance mechanisms to ensure its availability and reliable communication.

2.3 Protocol Outline

F-PReP[4] has been designed to meet the following requirements:

– *Guaranteed Recording.* After a process finishes execution, the entire documentation of that process must eventually be recorded in provenance store(s).

[1] The provenance store has been implemented as a stateless web service with a database storage system. Hence the latest consistent state refers to the initial state of the service and the latest checkpointed state of the database.

- *Viewlink Accuracy.* Viewlinks recorded for each interaction of a process must eventually be accurate. Each must point to the store where the other actor in the same interaction recorded p-assertions documenting that interaction.
- *Causelink Accuracy.* Causelinks recorded during a process must eventually be accurate. Each must point to the store where p-assertions about the corresponding cause interaction were recorded.
- *Efficient Recording.* Recording p-assertions and taking remedial actions should be efficient and introduce minimum overhead.

Fig. 3 demonstrates an example of message exchanges in F-PReP. The default provenance stores that the sender and receiver use are $PS1$ and $PS2$, respectively. The sender and receiver create p-assertions documenting the interaction where an application message app is exchanged (Step 1). Asynchronously, they submit all their p-assertions about the interaction and a viewlink[2] in a single message, record, to their provenance store. Before delivering a record message, an actor checks all the relationship p-assertions in the message and updates incorrect causelinks in order to meet *Causelink Accuracy* requirement.

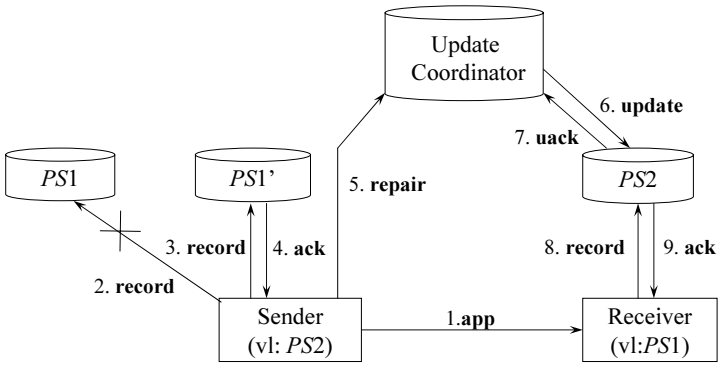

Fig. 3. An example of message exchanges in F-PReP

An actor sets a timeout when waiting for an acknowledgement ack immediately after it sends a record to a provenance store (Steps 2, 8). A provenance store acknowledges a record by means of an ack message, only *after* it has successfully recorded the content of record in its persistent storage. If an ack is not received before a timeout, an actor can conclude that failures may have occurred; it can then resend the same record to the actor's default store or an alternative store (Step 3). However, the use of an alternative store leads to incorrect causelinks or viewlinks, hence requiring an update. In the example of Fig. 3, the receiver's viewlink to $PS1$ becomes incorrect.

We introduced an update coordinator to facilitate viewlink updating. An update coordinator is necessary since both sender and receiver may issue a repair

[2] We assume the sender and receiver know their viewlink to $PS2$ and $PS1$, respectively, by means of built-in knowledge.

request in an interaction. This cannot be achieved by direct update of the other actor's provenance store, because at that moment, one does not know which store the opposite actor is actually using. In Fig. 3, the sender requests the coordinator (Step 5) to help update the receiver's viewlink in $PS2$ (Step 6). After updating a viewlink, $PS2$ returns an acknowledgement message uack (Step 7).

We assume *the update coordinator does not crash*. We can use the traditional fault-tolerance mechanisms such as replication to ensure its availability. This is feasible since a coordinator maintains only a small amount of information, as illustrated later. However, it is not feasible to replicate provenance stores which usually maintain a large amount of process documentation. Because replication, although sophisticated, comes with a significant cost due to the preservation of the *one-copy equivalence* property [11].

3 Implementation

The implementation of F-PReP involves three parts: Provenance Support Library (F-PSL), Provenance Store (PS), and Update Coordinator.

F-PSL includes a set of Application Programming Interfaces to create and record p-assertions into a provenance store. F-PSL extends the PReP-oriented PSL with the following novel functionalities:

(1) Remedial actions that cope with failures. First, F-PSL resubmits record messages according to the policies specified by a configuration file, including a list of alternative stores. Second, it maintains a history of the use of alternative stores during an actor's participation in a process. This information is currently maintained in memory and deleted when an application completes. Third, F-PSL checks an actor's causelinks when recording relationship p-assertions and updates them according to the history information. Fourth, F-PSL requests a coordinator to update viewlinks.

(2) Multithreading for the creation and recording of p-assertions. F-PSL enables the concurrent creation and recording of p-assertions during application execution. An application's requests for creating p-assertions are queued to be processed by a creation thread. Created p-assertions are also kept in a queue (in the form of record messages) before being submitted to a store by a recording thread. Basic flow control is provided in the form of queue management.

(3) A local file store for temporarily maintaining p-assertions. If recording p-assertions significantly degrades an application's performance, p-assertions can be maintained locally and submitted later. We employ Berkeley DB Java Edition database (BDB) as the local file store for its ease of installation.

PS has been implemented as a Java Servlet and deployed in the Apache Tomcat Servlet container[1]. It supports several types of backend data stores. Our implementation and experiments were based on BDB. We extend the current implementation of PS in terms of the following aspects:

(1) Disk cache. A PS *persistently* caches a received record message to disk before providing an acknowledgement[3], and at a later stage processes the record message and stores p-assertions. This caching mechanism delays actual message processing and hence saves on the overhead of processing messages.

(2) Update Plug-In. PS has been designed to facilitate convenient integration of new features through the use of plug-ins. A new plug-in, Update Plug-In, is implemented to receive update requests from the coordinator and update requested view links.

Update Coordinator is implemented as a Java Servlet and deployed in the Tomcat container. It receives a repair request from an actor and maintains the requested information in a local file store before sending an update message to a provenance store to update a requested view link.

Only minimum information is maintained for each repair request: the identity of the destination store that needs to be updated, and the identity of the store that successfully recorded the requesting actor's record message for a given interaction. The maintained information is used to cope with the case where both actors in an interaction request to update the other's viewlink in that interaction. The internal behavior of the coordinator and the management of maintained request information are detailed in [4].

Similarly to PS, the coordinator persistently caches received repair requests in its local file store, and at a later stage processes these requests to save on the overhead of processing messages. An actor continues its execution after receiving a response indicating its repair request has been cached in the coordinator. BDB is also employed as the file store.

An application can utilize multiple coordinators in its process. When using more than one, any two actors exchanging an application message must share the same one in order to ensure Viewlink Accuracy requirement. The identifier of a coordinator can be built in actors or exchanged to other actors in the application message app.

4 Performance Evaluation

Our experiments were run on the Iridis Computing Cluster at the University of Southampton. Iridis contains several sets of nodes (i.e. computers). Nodes used in the experiments each have two Single Core AMD Opteron processors running at 2.2 GHz and 2 GB of RAM. Provenance store and update coordinator were run on nodes each with 4 Dual Core AMD Opteron processors running at 2.4 Ghz and 2 GB of RAM. In the experiments with failures, one coordinator was employed, installed on a node of the cluster. All nodes are connected by Gigabit Ethernet. All applications used in the evaluation were written in Java and were run using the Java 1.5.0 05 64-bit Server Virtual Machine.

[3] The PS in PReP, though caching a record message into BDB before replying an acknowledgement, does not force the message into disk by flushing operating system's buffers, thus having a risk of losing p-assertions if operating system fails.

Since failures are non-deterministic in nature and typically very hard to predict, a generator was used on an actor's side to inject random failure events, i.e, failure to submit a record message to a provenance store or failure to receive an acknowledgement from a provenance store before a timeout. The presence of such an event triggers an actor to take remedial actions. The generator generates a failure event based on a failure rate, i.e., the number of failure events that occur during a total number of recordings. Given a timeout, the generator postpones generating a failure event until the timeout expires. The advantage of using a failure generator is that it enables us to fully control the number of failures that may occur so as to investigate the correlation between recording performance and failure rate.

Our evaluation was conducted at three levels. First, we measured the throughput of the provenance store and update coordinator. We also investigated how the contention for the coordinator affects an actor's recording performance when the number of recording actors increases. Then, we benchmarked the recording performance of F-PReP without considering contention. Third, we investigated F-PReP's impact on the execution time of a scientific application. In each level, we performed two experiments: failure-free experiment and experiment with failures. A comparison with PReP was made in the failure-free experiment. In our evaluation, when we say recording p-assertions using F-PReP or PReP, we mean their respective client side library, F-PSL or PSL, was used, and the provenance store was configured with or without disk cache, respectively.

4.1 Throughput Experiment

Provenance Store. In Section 3, a *disk cache* mechanism is introduced as the default setup of a provenance store in F-PReP. This means the store forces every received record message into disk before providing an acknowledgement in order to maintain the durability of p-assertions. However, this mechanism may sacrifice a provenance store's throughput (i.e. the number of p-assertions accepted in a period of time).

We performed two failure-free tests with and without *disk cache* enabled, respectively. On each node, we created up to 16 threads (i.e., clients) recording 10k p-assertions at the same time. An MPI based test harness was used in the experiments to guarantee that all clients were run in parallel. Given that an experiment is allowed to use up to 32 nodes in the Iridis environment, we can have 512 clients simultaneously recording p-assertions into a provenance store. P-assertions are recorded with a new record message created for each p-assertion. All p-assertions were directly created and submitted to a provenance store without using threading.

Fig. 4 shows the results. In both setups, the provenance store's throughput levels off, where about 212,200 and 176,000 10k p-assertions are accepted in a 10 minute period in the setup *without disk cache* and *with disk cache*, respectively. This means a store's throughput decreases by 20% due to enabling disk cache.

Update Coordinator. We also measured the coordinator's throughput (i.e. the number of repair requests accepted in a period of time) with up to 512

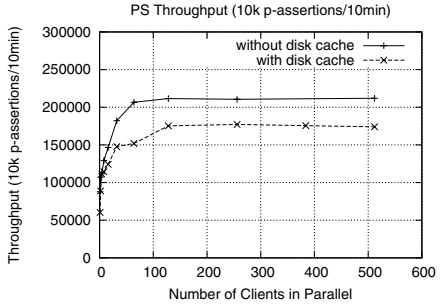

Fig. 4. Provenance Store Throughput

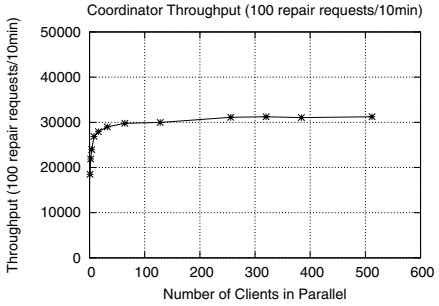

Fig. 5. Coordinator Throughput

clients simultaneously sending repair messages to an update coordinator. To save on the cost of network connection, 100 repair requests were sent to a coordinator in a single message. Fig. 5 shows that a coordinator can accept up to around 30,000*100 repair requests in a 10 minute period. This means there were 30,000*100 recording failures in 10 minutes, which is unlikely to see in applications.

4.2 Throughput Experiment with Failures

This experiment investigated: (1) the impact of contention for a coordinator on a client's recording performance[4]; (2) the tradeoff between resending a record message to the same provenance store and to an alternative store.

We conducted two experiments where a single client and 128 clients kept recording 10k p-assertions into one provenance store in a 10 minute period. Various failure rates (5%, 10%, 16%, 20%, 25%, 33% and 50%) were considered. We did not consider failure rates beyond 50% because it is not realistic [13]. Another provenance store was employed as the alternative store. One coordinator was used in the experiments and 100 repair requests were sent in a single batch. Since the more failures the more repair requests[5], failure events were immediately generated without considering timeouts to maximize the number of repair requests that could be sent to the coordinator within 10 minutes.

There exists a tradeoff between using the same provenance store or an alternative one when resubmitting a record message. Retransmitting messages to the same provenance store can tolerate transient failures, such as message loss. However, if a provenance store has crashed and is to be recovered after a long period of time, resending messages to the same store is not a good solution. On the other hand, the use of an alternative store, though provides guaranteed recording, ends up with an actor's causelinks or another actor's viewlink incorrect. This introduces additional cost of updating links. We compared the two approaches in our experiments.

[4] The impact of contention for a provenance store has been studied in [6].

[5] Recall that a repair request is produced after a record message is successfully recorded into an alternative store.

Fig. 6 shows the result in the experiment with a single client. The result was averaged from five runs of the experiment. We have two observations. First, when using the alternative store in each retransmission, up to about 20,000 repair requests are produced (because about 40,000 p-assertions are recorded when failure rate is 50%). This means the coordinator, in the worst case, receives 200 batches, each containing 100 repair requests, from a single client within 10 minutes. According to coordinator's throughput experiment in Section 4.1 and the fact that the 200 repair batches are received by the coordinator from a single client all across 10 minutes, we can imply that with about 100 clients, each having its own provenance store and alternative stores, the impact of contention for a coordinator on a client's recording performance would be very small.

The second observation is that resending messages to the same provenance store can record more p-assertions than to an alternative store, assuming that only transient failures are present. This is because the use of an alternative store requires extra actions to update links.

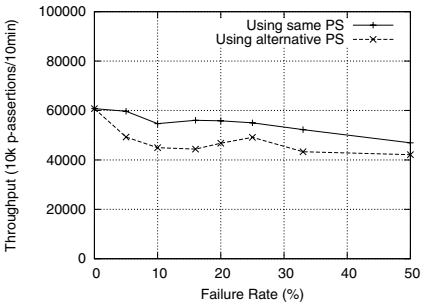

Fig. 6. Throughput experiment (single client)

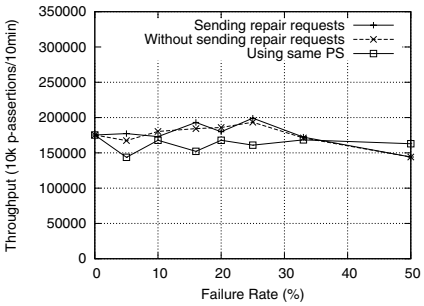

Fig. 7. Throughput experiment (128 clients)

Fig. 7 shows the result when 128 clients record p-assertions into one provenance store in the presence of failures. This experiment considers the contention for a provenance store as well as potential contention for a coordinator. We also have two observations. First, communicating with the coordinator does not affect total throughput. This implies that the contention for a coordinator is negligible (It can be calculated that up to about 750 repair batches are sent to the coordinator from 128 clients in 10 minutes.). Second, using an alternative store, in general, results in more p-assertions recorded than using a same store to resend p-assertions. This is because the use of an alternative store helps to balance the load of recording p-assertions (especially when failure rate is 25%), though introducing additional cost of updating links.

From these experiments, we have two conclusions. First, the coordinator is scalable and the impact of its contention on a client's recording performance is very small or negligible. Since our implementation supports the use of multiple coordinators, we believe the introduced component, update coordinator, does not affect an application's recording performance. Second, to achieve a better

recording performance, an alternative store should be employed after resending messages to a same provenance store has failed for certain times.

4.3 Benchmark Experiments

We now investigate the recording performance of a single actor without considering contentions. All the benchmark experiments were run with one client recording p-assertions into one provenance store. All p-assertions were directly created and submitted to a provenance store without using threading.

Failure-free Experiment. The experiment compares F-PReP to PReP in a failure-free environment. We measured the time to record 10,000 10k p-assertions. To minimize the impact of network connection overhead, 100 p-assertions were shipped in a same record message. Measurements were taken after recording a record message. Fig. 8 summarizes the record time. The graph displays an average from ten trials. From the figure, we have two observations:

(1) The provenance store without using disk cache, i.e., in the setup *using PReP*, periodically flushes 900 p-assertions from its operating system buffers into disk. This means if the provenance store's operating system crashes, up to 900 10k p-assertions may be lost.

(2) The average time to record 100 10k p-assertions is 198.8ms and 174.4ms using F-PReP and PReP, respectively. Therefore, F-PReP has an overhead of 13.8% compared to PReP. We note that in an application, the impact of F-PReP on the application's performance is similar to that of PReP, as illustrated later in the application experiment. This similarity benefits from the use of multithreading to asynchronously record p-assertions.

Experiment with Failures. In Section 4.2, we measured a client's recording performance in the presence of failures in terms of throughput. However, we did not consider the overhead of updating causelinks. Updating causelinks matters only when a relationship p-assertion is to be recorded. In this experiment, we

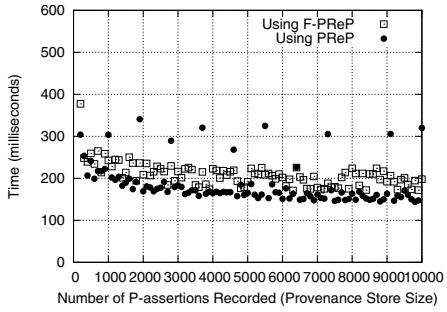

Fig. 8. Time to record 100 10k p-assertions

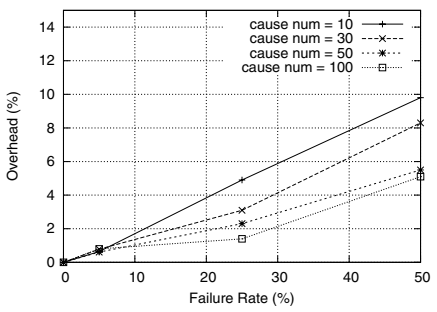

Fig. 9. Overhead of taking remedial actions

approximated the maximum overhead of taking remedial actions by measuring the record time of relationship p-assertions.

In F-PReP, the more causes a relationship p-assertion has, the longer it takes to check and update causelinks. Therefore, we increased the number of causes from 10 to 100.

Given a number of causes, several tests were conducted with various failure rates (5%, 25% and 50%). For each failure rate, the p-assertions about cause interactions of a relationship p-assertion were recorded prior to measuring the recording time for the relationship p-assertion itself. In order to measure the actual cost of remedial actions by means of record time, failure events were immediately generated without considering timeouts. We deployed another store as an alternative store, which was used in the retransmission of a relationship p-assertion. Repair requests were sent to a coordinator in batch sizes of 100.

Fig. 9 summarizes the results in terms of overhead. The measurements were taken after recording 100 relationship p-assertions. We can observe a maximum overhead of 10% for taking remedial actions, when compared to the record time when no failure occurred. Broadly speaking, the overhead increases linearly with the increase in failure rate. We note that since it takes much longer time to record a relationship p-assertion with larger number of causes, the overhead of taking remedial actions becomes relatively small in the settings with more causes. Therefore, we observe the smallest overhead in the setting with 100 causes.

4.4 Application Experiment

This experiment aims to investigate F-PReP's recording performance in a scientific application, the Amino Acid Compressibility Experiment (ACE), which has been detailed in [6]. ACE attempts to find possible new relationships between amino acids by investigating the information theoretic properties (e.g., information efficiency) of their computational representations.

ACE is chosen because of its general properties representing a range of workflow applications. First, it can be used to answer a range of provenance queries. Second, it is high performance and fine grained, which implies that p-assertion recording may be difficult. Therefore, the evaluation result obtained from this difficult application can imply a worst case complexity of that obtained from a large set of applications with less demanding requirements.

One run of ACE consisted of 20 parallel jobs[6]. Each job involved 54, 000 interactions between seven actors[7] in order to produce 4,500 information efficiency values. Actors used five provenance stores to record process documentation and these provenance stores were also employed as the alternative stores known by each actor. The process documentation created by ACE was extremely detailed;

[6] There is no dependency between jobs.

[7] A local method is instrumented as a recording actor using F-PReP. Actors exchange messages by means of method calls without network connections. They record p-assertions documenting the messages they receive and send to contribute to the process documentation of an information efficiency value.

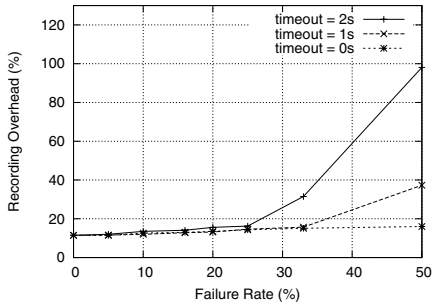

Fig. 10. Recording overhead of F-PReP

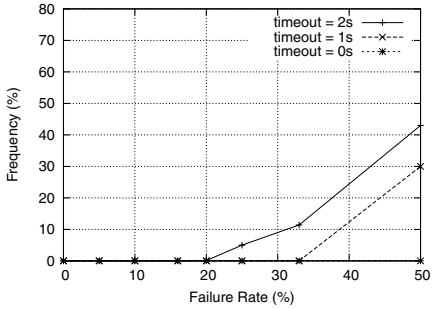

Fig. 11. The frequency of a queue in full capacity

the steps used to compute each result were recorded. The recorded process documentation could effectively answer all the use case questions in [6].

Each job produced 108,000 record messages, each containing about 10Kb p-assertions on average. To minimize network connection overhead, both record messages and repair requests were sent in batches of 100 each. Multithreading for creation and recording p-assertions was used in all tests. Various failure rates (5%, 10%, 16%, 20%, 25%, 33% and 50%) were considered. When taking remedial actions, a randomly selected alternative store was used in each resubmission.

We also investigated the impact of timeout on an application's performance. We studied three timeouts, 0s, 1s and 2s. The timeout, 0s, provides an extreme case, where a failure event occurs (or is detected) very quickly.

The application runtime is the average of the runtime of all parallel jobs from five runs of ACE. The runtime of an application without recording p-assertions is 22:24 (in the format *mm*:*ss*). When no failure occurs, the application runtime using PReP and F-PReP are 24:58 and 25:07, respectively. Therefore, the recording overheads of PReP and F-PReP are similar (about 12%). This benefits from the use of multithreading to asynchronously record p-assertions[8].

The asynchronous approach allows an application's p-assertions to be queued before being shipped to a provenance store. F-PReP has provided a flow control mechanism in queues to avoid exhausting memory. P-assertions cannot be queued until there is space in the queue. This may however affect the application's performance, since the application is postponed occasionally in order to reduce the speed of creating p-assertions when the queue becomes full frequently.

Our results in Fig. 10 and Fig. 11 demonstrate the correlations among application performance, failure rate, timeout and queue utilization. In Fig. 10, the recording overhead slightly increases as the failure rate increases in all timeout setups. However, it is sharply increased at certain points. Fig. 11 shows how often a queue is in a full capacity when a new batch of record messages is to be enqueued. It clearly reveals that the sharp increase in the recording overhead in Fig. 10 results from the flow control mechanism.

[8] Multithreading was also used in the tests of PReP.

From this application experiment, we can draw several general conclusions:

(1) Both PReP and F-PReP have similar recording overhead when there is no failure (around 12% in ACE);
(2) If the recording queue's size is large enough, F-PReP introduces a small recording overhead in the presence of failures (below 20% in ACE);
(3) The timeout for receiving an acknowledgement from a provenance store can affect an application's performance. An appropriate timeout should be chosen.
(4) By monitoring the utilization of queues, we can detect if an application's performance has been severely degraded and then take actions to improve the performance. For example, the local file store introduced in F-PSL can be automatically employed for temporarily maintaining p-assertions[9], when the frequency of the queue in maximum capacity reaches a certain threshold, e.g., 40%.

Query. After each run of ACE in the presence of failures, we also queried the provenance stores to further verify the quality of documentation recorded by F-PReP. The results showed an equal number of documentation records in the stores and records produced in ACE. In addition, no isolated island or dangling link is found, and distributed documentation of the process that led to an information efficiency value can always be retrieved in its entirety.

5 Related Work and Conclusion

Several provenance frameworks have emerged in the past a few years, e.g., Karma [12], PASOA [6]. Some workflow systems also provide provenance collection functionalities, e.g., Kepler [2]. From an analysis of these works, the issue of recording process documentation in the case of failures has not been discussed. Xu et. al. [15] have proposed a framework to tolerate failures occurring in service-oriented systems. Their approach relies on provenance information recorded in the presence of failures, which would benefit from F-PReP.

There is not much work on performance study related to provenance. Performance evaluations of PReP are presented in [7,6]. A detailed comparison on recording and querying performance between Karma and PReServ is seen in [12]. Extensive performance evaluations have been made on techniques to reduce the amount of storage required for process documentation [3]. There has been a performance study on PASS [10], an automatic provenance collection and maintenance storage system at the operating system level. None of these evaluations considers failures.

In this paper, we have evaluated a protocol, F-PReP, for recording process documentation in the presence of failures. In a failure-free environment, it has similar impact on an application's performance as PReP does. Although it introduces overhead in the presence of failures, we believe the overhead is still acceptable given that it can record *high quality* process documentation.

We are currently investigating how to create process documentation when an application has its own fault tolerance schemes to tolerate application level

[9] When using a local file store, the recording overhead was about 42% in our test.

failures. In future work, we plan to make use of the process documentation recorded in the presence of failures to diagnose failures.

References

1. Apache tomcat. User guide, `http://tomcat.apache.org/tomcat-5.5-doc/index.html`
2. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance collection support in the kepler scientific workflow system. In: Moreau and Foster [9], pp. 118–132
3. Chapman, A., Jagadish, H.V.: Efficient provenance storage. In: SIGMOD Conference (June 2008)
4. Chen, Z., Moreau, L.: Recording process documentation in the presence of failures. In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Methods, Models and Tools for Fault Tolerance. LNCS. Springer, Heidelberg (accepted, 2008)
5. Gagliardi, F., Jones, B., Grey, F., Bgin, M.E., Heikkurinen, M.: Building an infrastructure for scientific grid computing: Status and goals of the egee project. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 363(1833), 1729–1742 (2005)
6. Groth, P.: The origin of data: Enabling the determination of provenance in multi-institutional scientific systems through the documentation of processes. Phd thesis, University of Southampton (2007)
7. Groth, P., Miles, S.: Weijian Fang, S. C. Wong, K.-P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In: Proceedings of 14th IEEE International Symposium on the High Performance Distributed Computing (HPDC), pp. 201–208 (2005)
8. Kloss, G.K., Schreiber, A.: Provenance implementation in a scientific simulation environment. In: Moreau and Foster [9], pp. 37–45
9. Moreau, L., Foster, I. (eds.): IPAW 2006. LNCS, vol. 4145. Springer, Heidelberg (2006)
10. Muniswamy-Reddy, K.-K., Holland, D.A., Braun, U., Seltzer, M.I.: Provenance-aware storage systems. In: USENIX Annual Technical Conference, General Track. USENIX, pp. 43–56 (2006)
11. Ozsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (1999)
12. Simmhan, Y.L., Plale, B., Gannon, D., Marru, S.: Performance evaluation of the karma provenance framework for scientific workflows. In: Moreau and Foster [9], pp. 222–236.
13. Tierney, B., Schopf, J.: The cedps troubleshooting architecture and deployment on the open science grid. Journal of Physics: Conference Series 78 (2007)
14. Wootten, I., Rajbhandari, S., Rana, O.F., Pahwa, J.S.: Actor provenance capture with ganglia. In: CCGRID, pp. 99–106 (2006)
15. Xu, J., Townend, P., Looker, N., Groth, P.: Ft-grid: a system for achieving fault tolerance in grids. Concurrency and Computation: Practice and Experience 20(3), 297–309 (2008)