Utility-Driven Availability-Management in Enterprise-Scale Information Flows

Zhongtang Cai[†], Vibhore Kumar[†], Brian F. Cooper[†],
Greg Eisenhauer[†], Karsten Schwan[†], Robert E. Strom[*]
[†]College of Computing,
Georgia Institute of Technology,
Atlanta, GA 30332
{ztcai, vibhore, cooperb, eisen, schwan}@cc.gatech.edu
[*]IBM T. J. Watson Research Center
Hawthorne, NY 10532
robstrom@us.ibm.com

April 5, 2006

Abstract

Enterprises rely critically on the timely and sustained delivery of information, supported by middleware that ensures high-availability for such information flows. Our goal is to augment such middleware to create resilient information flows that deliver information while maximizing the utility end user applications derive from such information. Towards this end, this paper presents a 'proactive availability-management' technique to offer (1) information flows that dynamically self-determine their availability requirement based on high-level utility specifications, (2) flows that can trade recovery time for performance based on the 'perceived' stability and failure predictions (early alarm) for the underlying system, and (3) methods, based on real-world case studies, to deal with both transient and non-transient failures. We have incorporated 'proactive availability-management' into information flow middleware, and experiments reported in this paper demonstrate its capability to self-determine availability guarantees, to offer improved performance over a statically configured system, and to be resilient to a wide-range of faults.

1 Introduction

Modern enterprises rely critically on timely and sustained delivery of information. The delivery of such information is often facilitated by distributed information flow middleware that acquires, manipulates, and disseminates information across the enterprise. An important attribute of companies' operational information systems is their availability - 24 hours a day, 7 days a week. Systems failures can have dire consequences for the enterprise, including loss of productivity, unhappy customers, or serious financial implications. In fact, the average cost of downtime for financial companies, as reported in [18], is up to 6.5 million dollars per hour and hundreds of thousands of dollars per hour for retail companies. This has resulted in strong demand for operational information systems that are available almost continuously.

Providing high availability in widely distributed operational information systems is complex for multiple reasons. First, because information flows are distributed, they are difficult to manage, and failures at any of a number of distributed components can reduce availability. Second, multiple flows may use the same distributed resources, increasing the complexity of the system and the difficulty of managing and preventing failures. Third, such systems often have high data rates and intensive processing requirements, and there are frequently not enough system resources to replicate all this data and processing to achieve high reliability. Fourth, information flows must have negligible recovery times to limit losses to the enterprise. Finally, based on our experience working with industry partners like Delta Air Lines and Worldspan, systems must recover not only from transient failures but also from non-transient ones (e.g., failures that will recur unless some root cause is addressed) [14].

How can we provide high availability for information flows, given all of these requirements? Traditional techniques such as recovery from disk-based logs [15] may have recovery times that are unacceptable for the domain in question. Using active replicas [26] imposes high communication and processing overheads (since all data flow and processing is replicated) and therefore may not be an economically viable option for the enterprise. Another option is to use an active-passive pair [26], where a passive replica of an operator can be brought up to date by retransmitting messages that had gone to the failed, active node. This option reduces communication costs, since messages are only sent to the passive node at failure time. Unfortunately, this means that the recovery time might be quite long. The ideal solution would be some hybrid of the above approaches, since we might be willing to spend more processing and communication during normal operation if it minimized recovery time.

We extend the active-passive approach to allow us to tune the tradeoff between normal operation cost and recovery time. In particular, the passive replica will be periodically refreshed with "soft-checkpoints:" these checkpoints transfer the current state from the active node to the passive node, but are not required for correctness (hence, they are "soft"). If the passive replica has been recently brought up to date by a soft-checkpoint, the recovery will be relatively fast. By changing the frequency at which soft-checkpoints are transmitted during normal operation, we can tune the tradeoff between cost and recovery time.

In this paper, we propose self-adaptive techniques for tuning the soft-checkpoint interval (and hence the cost/recovery time tradeoff). These techniques manage the availability of multiple information flows in order to achieve the highest overall value for the enterprise. Our system provides:

- *Availability-Aware Self-Configuration* – a user-supplied per information flow 'benefit-function' drives the assignment of the resources required to guarantee availability. This ensures preferential treatment of flows that offer more benefit to the enterprise, with the aim of maximizing benefit across the system.

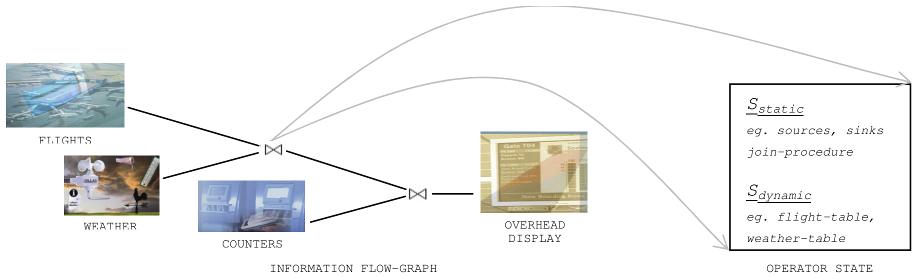


Figure 1: Information Flow-Graph and Operator State

- *Proactive Availability-Management* – during its execution, a system may be at different levels of stability (e.g., a heavy memory load could mean an imminent failure). In many cases, the "current stability" of the system can be quantified in order increase or decrease the resources expended to ensure desired levels of availability.
- *Handling Non-Transient Failures* – Some failures will recur if the same sequence of messages that caused the failure is resent during recovery. In this case, we must use application-level knowledge to avoid fault recurrence. We present generic techniques, based on real-world case studies, to deal with such faults.

Our techniques have been integrated into IFLOW, a high performance information flow middleware we have previously developed [20, 21], to provide a coherent infrastructure for managing the availability of large-scale information flows. Experimentally, we show that proactive availability-management imposes low additional communication and processing overheads. Experiments with IFLOW deployed on Emulab [22] also demonstrate the effectiveness of proactive fault tolerance in recovering from failures, in offering a low recovery time of 2.5 seconds for a representative emulated enterprise-scale information flow, while simultaneously, offering 1.5 times the net-utility when compared to the active replica approach. In contrast to our previous work on IFLOW, this paper focuses on utility-driven fault tolerance, using IFLOW as the means to realize and experiment with the concept.

1.1 Example: Operational Information System

An operational information system (OIS) [14] is a large-scale, distributed system that provides continuous support for a company or organization's daily operations. One example of such a system we have been studying is the OIS run by Delta Air Lines, which provides the company with up-to-date information about all of their flight operations, including crews, passengers and baggage. Delta's OIS combines three different sets of functionality:

- *Continuous data capture* - for information like crew dispositions, passengers, airplanes and their current locations determined from FAA radar data.
- *Continuous status updates* - for low-end devices like airport flight displays, for the PCs used by gate agents, and even for large databases in which operational state

changes are recorded for logging purposes.

- *Responses to client requests* - an OIS must also respond to explicit client requests such as pulling up information regarding a particular passenger and may also generate additional updates for events such as changes in flights, crews or passengers.

In this paper, we model the information acquisition, manipulation and dissemination done by the OIS as an information-flow graph (an example flow-graph is shown in Figure 1). We then present techniques, based on this flow-graph formalization, to proactively manage the availability of the operational information system such that the net-utility achieved by the system is maximized. This is done by assigning per information-flow availability guarantees that are aligned with the benefit that is derived from the information flow, and by proactively responding to the perceived changes in system stability. We also present additional techniques, based on real-world case studies, that can help a system recover from non-transient failures.

2 System Overview

This section describes a formal model of the information flows under consideration, and the fault model used for the proactive availability-management technique explained later.

2.1 Information Flow Model

An information flow is represented as a directed acyclic graph $G(V_g, E_g, U_{net})$ with each vertex in V_g representing an information-source, an information-sink or a flow-operator that processes the information i.e. $V_g = V_{sources} \cup V_{sinks} \cup V_{operators}$. Edges E_g in the graph represent the flow of information, and may span multiple intermediate edges and nodes in the underlying network. Finally, the utility-function U_{net} is defined as:

$$U_{net} = Benefit - Cost \qquad (1)$$

Both benefit and cost are expressed in terms of some unit of value delivered per unit time (e.g., dollars/second). Benefit is a user-supplied function that maps the delay, availability, etc. of the information flow to its corresponding value to the enterprise. Cost is also a user-supplied function, and maps resources such as CPU usage and bandwidth consumed to expense incurred by the enterprise. We will expand the terms of this seemingly simple equation in upcoming sections.

2.2 Fault Model

We are concerned with failures of the information flow that occur after it has been deployed. In particular, we focus on fail-stop failures of operators that process events. Such failures could result from problems in the operator code or in the underlying physical node. Other factors might also cause failures, but are not considered here, including problems with sources, problems with the sink, or link failures between nodes. While such issues can cause user-perceived failures, they are outside the control of our middleware and thus, must be addressed using other techniques. For example, link failures could be managed by retransmission or re-routing at the network level.

For the purpose of recovering from a fault, we assume that each flow-operator consists of a *static-state* S_{static}, that contains the information about the edges connected to the operator and the enterprise logic embedded in the operator; in contrast, the *dynamic-state* $S_{dynamic}$, is the information that is a result of all the updates that have been processed by this operator (shown in Figure 1). Recovery therefore is dependent upon the correct retrieval of the states S_{static} and $S_{dynamic}$, which together contain the information necessary for re-instantiation of flow-operator and information flow edges. However, as we describe next, simply recovering these states may not prevent the recurrence of a failure.

2.2.1 Transient Faults

A fault can be caused by a condition that is transient in nature (e.g., a memory overload due to a mis-behaving process), and such faults will not recur once the system starts again after a recovery. A transient fault in our formalization would cause the failure of an operator, and correct retrieval of the two states associated with the operator would ensure permanent recovery from this fault. The techniques proposed in this paper are capable of effectively handling a fault of this nature.

2.2.2 Non-Transient Faults

Non-transient faults are generally caused by some bug in the code, or due to some unhandled conditions. For information flows, this would mean recurrence of the fault even after recovery, if the same sequence of messages that caused the fault are simply repeated. To deal with faults of this nature, we note that the output produced by a flow-operator in response to an input event E, depends on the existing dynamic-state $S_{dynamic}$, the operator logic encoded as S_{static}, and the event E itself. Therefore, the failure of an operator on arrival of an event E, is a result of three tuple $< S_{dynamic}, S_{static}, E >$. Thus, any technique that aims to deal with non-transient failures must have application-level hooks with which to retrieve and appropriately modify this 3-tuple. Our prior work presents some examples of such hooks [14].

3 Utility-Driven Proactive Availability-Management

Traditional techniques for availability-management typically rely on undo-redo logs, active-replicas, or an active-passive pair. A new set of problems is presented by information flows that form the backbone of an enterprise. For instance, using traditional on-disk undo-redo logs for information flows would lead to unacceptable recovery times for the enterprise domain in face of machine or disk failures. The other end of the availability-management spectrum, which uses active replicas, would impose large additional communication and processing overheads due to the high arrival rate of updates, making it economically infeasible for the enterprise to entertain this option. In response, researchers have customized the active-passive pair algorithm [26] for enterprise-scale information-flows. This customization uses our novel notion of 'soft-checkpoints', first presented in [32], to reduce communication and processing overheads. The basic method and its customization are described next.

3.1 Basic Active-Passive Pair Algorithm

To ensure high-availability for the flow-operator, in its simplest form, the active-passive pair replication requires: a *passive node* containing the static-state S_{static} of the flow-operator hosted on the active node, an *event log* at the flow-graph vertices directly upstream to the flow-operator in question, a mechanism to *detect duplicates* at the vertices directly downstream to the flow-operator, and a *failure detection* mechanism for the active node hosting the primary flow-operator.

In case of a failure, recovery proceeds as follows: the failure detection mechanism detects the failure and reports the same to the passive node. On receipt of the failure message, the passive node instantiates the flow-operator, making use of the static-state S_{static}, already available at the node. The instantiated operator then contacts the upstream vertices for re-transmission of the events in their event log. The newly instantiated operator node processes these re-transmitted events in a normal fashion, generating output events, and leaving it to the downstream nodes to detect the resulting duplicates. Once the re-transmission of the event log has been completed, the resulting dynamic-state, $S_{dynamic}$, will be recovered to the state of the failed operator, and normal operations can resume. Unfortunately, this simple algorithm can lead to high recovery times, large event logs at the upstream nodes, and large associated re-transmission costs. The remedy to these problems is the 'soft-checkpoint' technique, described next.

The event logs at the upstream nodes and their re-transmission to the recovered operator are required for reconstructing the dynamic-state $S_{dynamic}$, of the failed operator. However, in practice, it is advantageous to retain additional stable state at the passive node in order to avoid the need to re-transmit the entire event log. Such state saving is called soft-checkpointing, because it is not needed for correctness. Soft checkpoints can be updated on an intermittent basis in the background. Once taken, the component receiving the checkpoint no longer requires the events on which the state depends for reconstructing $S_{dynamic}$. This in turn permits upstream nodes to discard the event logs for which the soft-checkpoint has been taken. Soft-checkpointing, therefore, is an optimization that reduces the worst-case recovery time and permits the reclamation of logs.

The introduction of soft-checkpoints requires small modifications to the recovery mechanism described earlier in this section. The flow-operator at the active node in the duration prior to failure would intermittently send messages to the passive node that contain information about the incremental change to its dynamic-state since the last message. The passive node, after the receipt of complete state update message from the active node, applies the incremental modifications to the state it holds and sends a message to the flow-operator's upstream neighbors about the most recent event contained in the message from the active node. The upstream nodes can use such information to purge their event logs. In case of a failure, the algorithm proceeds exactly as described earlier, but only a small fraction of the events need to be re-transmitted and processed.

3.2 Availability-Utility Formulation

In this section, we use a basic availability formulation to better describe the effects and trade-offs in soft-checkpoint-based active-passive replication. Availability $\mathcal{A}_{\mathcal{I}}$ is described in terms of Mean-Time Between Failure, $MTBF$ and Mean-Time To Repair, $MTTR$.

$$\mathcal{A}_{\mathcal{I}} = \frac{MTBF}{MTBF + MTTR} \tag{2}$$

As stated earlier, our approach contributes to a reduction in recovery time and also reduces the processing and communication overhead imposed as a result of ensuring a certain level of availability. The reduction in recovery time results in lower MTTR, the reduction in associated overheads diminishes cost, and both together result in higher net-utility U_{net}, which is the actual utility provided by the system.

With our approach, MTTR depends on two factors: (1) the time to detect a failure, and (2) the time to reconstruct the dynamic-state of the operator. Failure detection mechanisms generally rely on time-outs to detect failures and therefore, depend on the coarseness of the timer used for this purpose. Some research in the domain of fault-tolerance has focused on multi-resolution timeouts [30], but to simplify analysis, henceforth, we assume that the time to detect a failure is a constant. The second factor contributing to MTTR depends on the soft-checkpoint algorithm. Specifically, a higher frequency f_{cp}, expressed in per unit time, of such checkpoints would lead to a smaller number of events required to reconstruct $S_{dynamic}$ in case of a failure. Therefore:

$$MTTR \propto \frac{1}{f_{cp}} \tag{3}$$

For simplicity, we next derive the availability-utility formulation for a single information flow (self-configuration across multiple information flows is addressed in Section 3.3), and we assume that the $Benefit$ and $Cost$ depend only on availability. In this case, in general, the benefit derived from a system is directly proportional to its availability. Thus:

$$Benefit \propto \frac{MTBF}{MTBF + k_1/f_{cp}} \tag{4}$$

The above formulation may lead one to believe that a higher f_{cp} is good for the system. Unfortunately, a higher f_{cp} also means more cost to propagate checkpoints from the active node to the passive node. Therefore:

$$Cost \propto f_{cp} \tag{5}$$

Note that a higher f_{cp} also results in fewer events retransmitted during recovery; however, for large values of MTBF this effect is minor compared to the effects described above (increase in benefit due to better availability, and increase in cost due to a higher frequency of checkpoints). Experiments reported in Section 5.2.4 study the effects of soft-checkpoint frequency on cost and availability of the information flow.

Combining equations 1, 4, 5, and replacing proportionality using constants, we arrive at:

$$U_{net} = \frac{k_2 \times MTBF}{MTBF + k_1/f_{cp}} - k_3 \times f_{cp} \tag{6}$$

This equation expresses the key insight that net-utility depends not only on MTBF, but also on the soft-checkpoint frequency used in a system, the latter both positively contributing to net-utility (by reducing the denominator) and directly reducing net-utility (by increasing the term being subtracted). Intuitively, this means that frequent checkpointing can improve utility by reducing MTBF, but that it can also reduce utility by using resources that would otherwise directly benefit the information-flow.

3.3 Availability-Aware Self-Configuration

Ideally, we would like to maximize the availability of an information flow, but given that there is an associated cost, our actual goal is to choose a value of availability that maximizes its net-utility. In our algorithm and its mathematical formulation f_{cp} is the factor that governs availability. By setting the derivative of equation 6 equal to zero, we find that the value of f_{cp} that maximizes net-utility is:

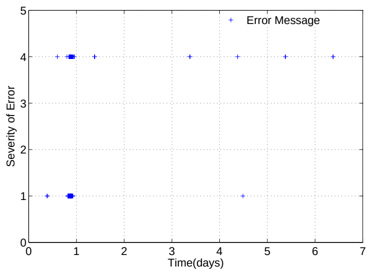


Figure 2: Enterprise error-log showing predictable behavior of failures

$$f_{cp} = \sqrt{\frac{k_1 \times k_2}{k_3 \times MTBF}} - \frac{k_1}{MTBF} \tag{7}$$

In the presence of multiple information flows, each with a different benefit-function, the resource assignment for availability is driven by the need to maximize net-utility across all deployed information flows. Total net-utility of the entire system, then, is the sum of individual net-utilities of information flows. For a system with n information flows, we will need to calculate $\{f_{cp}^1, f_{cp}^2, ..., f_{cp}^n\}$, which will automatically determine resource assignments. The value of f_{cp} for each information flow can be calculated using partial differentials, and the involved calculations are omitted due to space constraints.

3.4 Proactive Availability-Management

We have established that net-utility depends on checkpoint frequency and MTBF. However, the MTBF in a real system is not a constant. Instead, the rate of failures fluctuates, with more failures occurring when the system is in an unstable state. For example, during periods of extreme overload, the system is likely to experience many component failures. If we can better approximate the current MTBF, and in particular predict when there will be many failures, we can make better decisions about checkpointing, increasing the checkpoint rate when the current MTBF is low (and failures are imminent.)

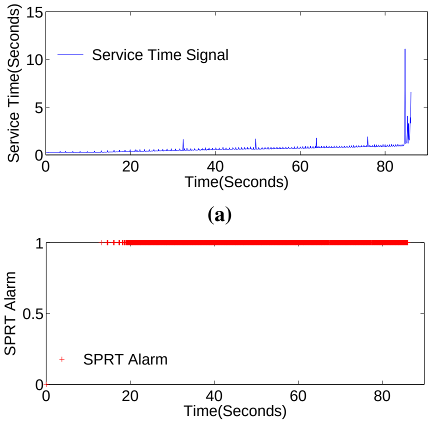
3.4.1 Failure prediction

An effective way to estimate the current MTBF is to use failure prediction techniques to generate 'early alarms' when a failure seems to be imminent. By using failure prediction methods, our approach can be 'better prepared' for an imminent failure, by taking more frequent soft-checkpoints. Analysis logs provided to us by one of our industry partners strengthens our belief in the usefulness of dynamic failure prediction. These logs contain error messages and warnings that were recorded at a middleware broker over a period of 7 days, along with their time-stamps. Figure 2 shows the distribution and severity of errors recorded at the broker node. One interesting observation of these logs is that errors recur at almost the same time (around 9:00am as read from the log time-stamp) beginning from the 2nd day. Another interesting observation about the same set of logs is that 128 errors of severity level one occurred from 7:30pm in the first day before a series of level four errors occurred from 8pm. Based on such logs, it would be reasonable, therefore, to assume lower MTBF (i.e., predict imminent failures) for the 9am time period and the period when large number of less severe errors occur, than for other time periods in which this application executes. We note that similar time- or load-dependent behaviors have been observed for other distributed applications.

We implemented the Sequential Probability Ratio Test used in MSET [34] failure prediction method. The sequential probability ratio test is a run-time statistical hypothesis test which can detect statistical changes in noisy process signals at the earliest possible time, e.g., before the process crashes, or severe service degradation occurs. As compared with normal threshold test, SPRT is capable of detecting failures much earlier than threshold test could. Meanwhile, as compared with standard fixed sample test in which a given number of observations are used to select one hypothesis from several alternative hypotheses, SPRT is capable of analyzing process observations sequentially at run-time and determine whether it has sufficient information to ensure pre-specified confidence bounds are met and if so, it can determine immediately whether or not the monitored process is consistent with normal behavior. SPRT has been applied successfully to the nuclear power plant on-line monitoring problem, and recently is been used for software aging problems, e.g., database lath contention problem, memory leak, unreleased file locks, and data corruption etc, with the benefits outlined above and also its high capability of achieving high sensitivity for subtle anomaly detection without increasing false alarm probability.

The SPRT method works in the following ways. During the initialization, user specifies the required false-alarm probability, missed-alarm probability, and the system disturbance magnitude. SPRT is trained during system normal operating period, so SPRT can learn the process signals which represent the usual operating state of the system. The duration of the training phase is relatively short, of the magnitude of several seconds to minutes, depending on the frequency of the signal. During the monitoring phase, SPRT records a sequence of the monitored signal and analyzes the signal with four hypothesis tests, the positive mean test H_1, the negative mean test H_2, the normal variance test H_3, and the inverse variance test H_4. Each test generates a SPRT index, which is compared to an upper limit and an lower limit(derived directly from the specified false-alarm probability and missed-alarm probability). If the lower limit is reached, the process is declared healthy, the test statistic is reset to zero(the previously recorded sequence of signal can be discarded now), and sampling continues. If the upper limit is reached, the process is declared degraded, an alarm is raised indicating process failure, the test statistic is reset to zero and sampling continues. If the neither of two the limits is reached, no decision can be made and the sampling continues. Readers who are interested in the mathematical formulations and other details, can refer to [16] for more details, and refer to [34] for variations of SPRT method.

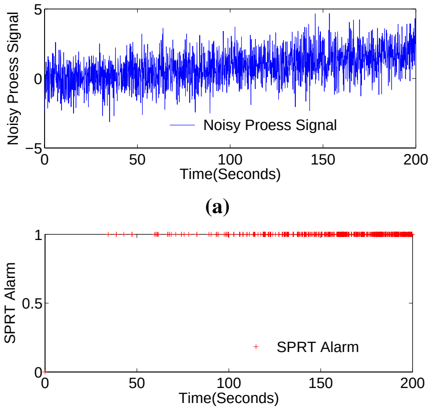
Figure 3 illustrates how SPRT can detect memory leak faults before the memory leak causes significant problem(slow response to no response from the process because of out of memory). Memory leak faults are injected at time $t = 10Sec$. Using heartbeat based failure detection, such kind of faults can only be detected when the all the memory is used up and the service degrades dramatically or the process crashes. In this example the service degrades dramatically at time $t = 45Sec$. However, SPRT is able to raise alarms starting from $t = 13.1Sec$. Such kind of early warning capability is reported in other literatures. For example, the eCM_{TM} system [6] reports early warning is raised from 5 minutes to 2 hours prior to database shared-memory-pool



(a)

(b)

Figure 3: SPRT alarms for memory leak failures



(a)

(b)

Figure 4: SPRT alarms for a synthesized noisy signal

latch contention failures. Figure 4 represents a more general and noisy signal studied in [16], in which linear service degradation starts from $t = 40Sec$. SPRT starts to raise alarms from time $t = 45Sec$. Note that simple threshold test can only detect this degradation much later.

One thing to note is that it is common sense that no failure prediction algorithm will work for all possible system failures [11], and the prediction accuracy could vary depending the algorithm and many other factors. One evidence is, although MSET and its SPRT can raise early alarms effectively in previous cases, there are situations in which false alarms are raised, or the internal fault causes the process failure or system-wide failure immediately and early alarm is almost impossible. For example, we use FIMD [3] software to inject software failures including timing delay, omission, message corruption datatype, message corruption length, message corruption destination, message corruption tag, message corruption data, memory leak, and invalid memory access. The invalid memory access normally cause process crash immediately and leaves almost no possibility for failure prediction/early alarm. Message corruption data fault sometimes would cause change in the input/output signal, while sometimes will not, depending on the internal service logic. In the latter case, no early alarm can be raised.

While there are several researchers working on improving the failure prediction mechanism, the focus of this paper is not to improve certain prediction algorithm, but rather to study if current prediction methods could help to improve the system availability, how the system could use such kind of prediction methods and also the accuracy requirement for the prediction algorithm to make it effective. Later we will also show that while the current prediction methods don't necessary provide very high accuracy for every kinds of failures, they do can help to achieve high availability, if proactive replication is 'regulated' in proper ways based on the prediction accuracy and replication cost etc. Meanwhile, interestingly enough, there are factors playing more important role in such kind of proactive system, which must be considered carefully to make early alarms really effective for enterprise-scale distributed systems.

3.4.2 Modulating the checkpoint frequency

The simple idea behind proactive availability-management is to use failure prediction to modulate f_{cp}; if a prediction turns out to be correct, the system 'benefits' because of reduced $MTTR$; if a prediction turns out to be a false-positive, the system still operates correctly, but it pays the 'cost' due to increased f_{cp}. Stated more formally, let:

$$\begin{aligned} \alpha &= \text{prediction false–positive rate} \\ \beta &= \text{prediction false–negative rate} \\ f'_{cp} &= \text{modulated checkpoint frequency after a failure is predicted} \\ T_{proactive} &= \text{duration of increased checkpoint frequency} \\ k &= \text{timeout after which an operator is concluded to be failed} \end{aligned}$$

Earlier, $Cost$ was shown to be proportional to soft-checkpoint frequency. The new cost, $Cost'$, due to modulated f'_{cp}, is:

$$Cost' = \frac{f'_{cp}}{f_{cp}} \times Cost \tag{8}$$

This increased cost is incurred for a duration equal to $T_{proactive}$, and it is incurred each time a prediction is made. Therefore, the additional cost incurred per prediction is:

$$\delta Cost = (\frac{f'_{cp}}{f_{cp}} - 1) \times Cost \times T_{proactive} \tag{9}$$

The increase in f_{cp} also affects the availability of the systems and therefore, the benefit, $Benefit'$ derived from the system. Using equation 4, we have:

$$Benefit' = \frac{MTBF + k_1/f'_{cp}}{MTBF + k_1/f_{cp}} \times Benefit \tag{10}$$

Therefore, the increase in benefit due to a correct prediction that affects a period equal to $MTBF$, is:

$$\delta Benefit = (Benefit' - Benefit) \times MTBF \tag{11}$$

Since λ is the fraction of false-positives and because there is no increase in benefit due to a false positive, the following condition expresses when proactive availability-management based on failure prediction is beneficial for an entire system:

$$\delta Cost < (1 - \alpha) \times \delta Benefit \tag{12}$$

Different systems could have different types and formulations of benefit and cost, and this analysis provides the most important insight and guideline. For the enterprise information flow system targeted by this paper, the proactive availability-management problem can be formulated in more details as followings. Proactive availability-management regulates the checkpoint frequency based on the stability predictions to minimize the total cost which in turn maximizes the net business utility. To simplify the analysis, the total cost here includes the cost of checkpointing and the utility loss because of the failure(i.e. the extra utility the system could offer if there is no failure), so the problem of maximizing net-utility is converted to the problem of minimizing the total cost. The total cost $Cost$ associated with proactive availability-management includes the cost of checkpoints $Cost^{cp}$, cost due to false-positive failure prediction(failure predictor raises a false alarm) $Cost^{fp}$, cost due to false-negative failure prediction(a failure is not predicted successfully) $Cost^{fn}$, and finally the cost associated with failure recovery when a failure is successfully predicted $Cost^{ps}$.

The cost of checkpoints $Cost^{cp}$ is given by:

$$Cost^{cp} = (1 - P(1 - \beta + \alpha))f_{cp}C_1, \tag{13}$$

where C_1 is the cost for each checkpoint update(e.g., the communication cost), and P is the possibility an operator could fail from any time t to t+1(second). In this equation, $P(1 - \beta + \alpha)$ is the fraction of one unit time when the checkpoint frequency is $f_{cp}^{'}$, due to correct failure predictions and false-positive predictions(false alarms).

The cost due to false-positive failure prediction is:

$$Cost^{fp} = \alpha P f'_{cp} t_o C_1, \tag{14}$$

where t_o is the average time a predictor can raise an early alarm for a severe failure before the failure actually causes the operator down.

The cost due to false-negative failure prediction $Cost^{fn}$ is:

$$Cost^{fn} = \beta P \frac{C_2}{2f_{cp}} + \beta P(k + \frac{1}{2f_{cp}})C_3, \tag{15}$$

where C_2 is the cost to recover the state update happened every unit time in the active operator. The first term is the cost for the passive node to recover from the last check pointed S_{static} to the S'_{static} when the failure occurred. The second term is the loss of utility when the system recover from this failure. In other words, this term represents the utility the system could provide if there is no such a failure.

The cost associated with failure recovery when a failure is successfully predicted, $Cost^{ps}$, is determined in a similar manner stated above:

$$Cost^{ps} = (1 - \beta)P[\frac{C_2}{2f'_{cp}} + (k + \frac{1}{2f\prime_{cp}})C_3 + f\prime_{cp} t_0 C_1]. \tag{16}$$

And the total cost is:

$$Cost = Cost^{cp} + Cost^{fp} + Cost^{fn} + Cost^{ps}. \tag{17}$$

To regulate the checkpoint frequency, proactive fault tolerance finds the best checkpoint frequency f_{cp} when there is no failure predicted and the checkpoint frequency f'_{cp} after failure is predicted, by minimizing the above formula.

Often, enterprises also has specific requirement of system availability. For example, a 365 x 24 system with maximum average downtime of 8.76 hours(525 minutes) per year requires 99.9 percent availability, while a system with only 3 minutes of service outrage must have at least 99.999 percent availability. Achieve such kind of availability is difficult due to the extremely high cost of fault tolerance services and equipments. Proactive availability-management strikes a good balance between these two factors by the process stated in the followings.

Notice that MTTR can be expressed as:

$$MTTR = \left(\frac{1}{2f_{cp}} + k\right)\beta + \left(\frac{1}{2f'_{cp}} + k\right)(1-\beta), \tag{18}$$

where k is the time out after which we conclude a module actually failed.
We immediately have:

$$\begin{aligned} A_I &= \frac{MTBF}{MTBF + MTTR} = \frac{1 - P \cdot MTTR}{1} \\ &= 1 - p[\left(\frac{1}{2f_{cp}} + k\right)\beta + \left(\frac{1}{2f'_{cp}} + k\right)(1-\beta)]. \end{aligned}$$

Proactive fault tolerance meets the minimum availability requirement and meanwhile maximize the net utility by solving the problem

$$Minimize\{Cost = Cost^{cp} + Cost^{fp} + Cost^{fn} + Cost^{ps}\}, \text{subject to:}$$

$$1 - p[\left(\frac{1}{2f_{cp}} + k\right)\beta + \left(\frac{1}{2f'_{cp}} + k\right)(1-\beta)] \geq A_I^{required}. \tag{19}$$

Normally, the utility function(net-utility) is a business-level specification, thus could vary in different enterprises. In Section 5.1, we also show experimental results with different utility function specification which is similar to the utility function in [21] to demonstrate the genrality and effectiveness of our approach.

3.5 Handling Non-Transient Faults

Non-transient failures are a result of bugs or unhandled conditions in operator code. Traditional techniques for ensuring high-availability that use undo/redo logs [15, 32] are useful when failures are caused by transient conditions. Using such techniques for non-transient failures would result in recurrence of faults during recovery. The same applies to replication-based approaches [2], in which all replicas fail simultaneously in face of non-transient faults.

As described in Section 2.2.2, a non-transient failure of the information flow in our model is a result of the three tuple $< S_{static}, S_{dynamic}, E >$. The active-passive pair approach for ensuring high-availability has enough information during the recovery to change this three tuple. The passive-node during recovery has access to S_{static}, a stale state $S'_{dynamic}$ and a set of updates T from the upstream nodes that when applied to $S'_{dynamic}$ would lead to $S_{dynamic}$. The rationale behind our approach to avoid non-transient failures is simple: avoid the three tuple that caused the failure. This can be done in a number of ways, and the retransmitted updates T along with the application-level knowledge hold the key:

- *Dropping Updates*: the simplest solution to avoid recurrence of a fault is to avoid processing the update that caused the failure. Our earlier work on 'poison messages' used this technique [14].

- *Update Reordering*: changing the order in which updates are applied to $S'_{dynamic}$ during recovery can avoid $S_{dynamic}$. This makes use of application-level knowledge to ensure correctness.
- *Update Fusion*: combining updates to avoid an intermediate state could be an option. A simple example of this approach could be the use of this technique to avoid 'division by zero' error.
- *Update Decomposition*: decomposing an update into a number of equivalent updates can be an option with several applications, and this can potentially avoid the fault.

While seemingly simple, the techniques described above are often successful in realistic settings. For example, one of our collaborators, Delta Airlines serving the Atlanta region, reported an occasional surge in the usage of resources connected to their Operational Information System (OIS) [23] that traced back to a particular uncommon message type. The resulting performance hit caused other subsystem's requests to build up, including those from the front ends used by clients, ultimately threatening operational failure (e.g., inappropriately long response times) or revenue loss (e.g., clients going to alternate sites). Such uncommon request/message termed as 'Poison Messages' were later found to be identifiable by certain characteristics. The solution then adopted was to either drop or re-route the poison message in order to maintain operational integrity.

4 Middleware Implementation

IFLOW [20, 21] is an information flow middleware developed at Georgia Tech. IFLOW implements the information flow abstraction of Section 2.1 and provides methods to deploy and then optimize (by migrating operators) the information flow. For more details please refer to [28].

We now briefly describe the features that enable proactive availability-management in the IFLOW middleware. These features are implemented both at the *control plane* and the *data plane* of this middleware infrastructure, and are described next.

4.1 Control Plane

The control plane in IFLOW is responsible for self-management of the information flow. This involves running a self-configuration and a self-optimization algorithm, carried out by exchange of control messages between physical nodes that are external to the data fast paths used to transport IFLOW data. Control actions involve operations like flow-control, operator re-instantiation, etc. The main features of the IFLOW control plane that are required for proactive fault tolerance are described below:

- *Availability-aware self-configuration module*: the benefit-formulation in IFLOW allows for availability goals to be specified, and determines the best value of f_{cp} by using the formulation described in Section 3.2.
- *Failure detection & prediction*: IFLOW attempts to use the regular traffic from a node to determine its liveness, but it switches to specific detection messages if there is no regular traffic from the node to the monitoring node. We also have a provision for multi-resolution timeouts to reduce the load imposed by the failure detection

algorithm. Finally, state can be maintained to use failure history for predicting failures, but we have not yet implemented any specific technique into IFLOW.

- *Control messages*: SOAP calls are used to notify active-node failure, to communicate log purge points to upstream vertices, etc.
- *Update re-direction in case of failure*: a simple control mechanism exists at the upstream vertices to re-direct updates to the passive node in case of failure. The connection between upstream vertices and the passive node is created at the time of flow deployment.

4.2 Data Plane

A *fast data-path* is one of the key design philosophies of the IFLOW middleware. We have taken care that the features required for proactive availability-management have minimal impact on the data-path. In order to ensure proactive availability-management, the state of an operator on the data plane needs to be soft-checkpointed and the changes need to be periodically communicated to the passive-node. The fact that a soft-checkpoint is not necessary for correctness of proactive availability-management ensures minimal impact on the data-path. Specifically, the active-node can transfer the soft-checkpoint to the passive node asynchronously (e.g., when load is low), and this will not compromise the correctness of our algorithm. The specific feature required for proactive availability-management are described below:

- *Logging at upstream vertices*: any update that is sent out from the source vertex is logged to enable retransmission in case of failure. Additional logs can be established at intermediate nodes (an operator vertex is a source for downstream vertices) to enable faster recovery. The log module also implements a mechanism to purge the log when a message is received from the downstream node after a soft-checkpoint is completed.
- *Soft-checkpoint module at operator vertices*: the soft-checkpoint module tracks the changes in $S_{dynamic}$ since the last soft-checkpoint. It is also responsible for sending soft-checkpoints to the passive node.
- *Duplicate detection at the downstream node*: the duplicate detection mechanism is based on the monotonic update system proposed in our earlier work [32]. When the updates cannot be ordered using the contained attributes, a monotonically increasing attribute (e.g., the real-time clock) is appended to the out-going update that uniquely identifies this update.
- *Additional edge between active-passive pair*: a supplementary data-flow between the active-passive pair delivers the soft-checkpoints to the passive vertex.
- *Maintaining checkpoints at passive-node*: the passive vertex contains the logic that applies an incoming soft-checkpoint to the recorded active node state.

5 Experiments

Experiments are designed to evaluate the performance our proactive availability-management techniques. First, simulations are used to better understand the behavior of the self-configuration module that determines the availability requirement based on the user

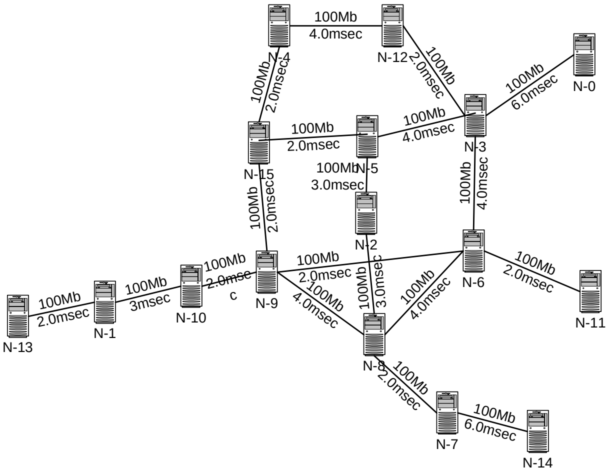


Figure 5: Sample testbed. The testbed topology is generated using GT-ITM and is configured at emulab facility.

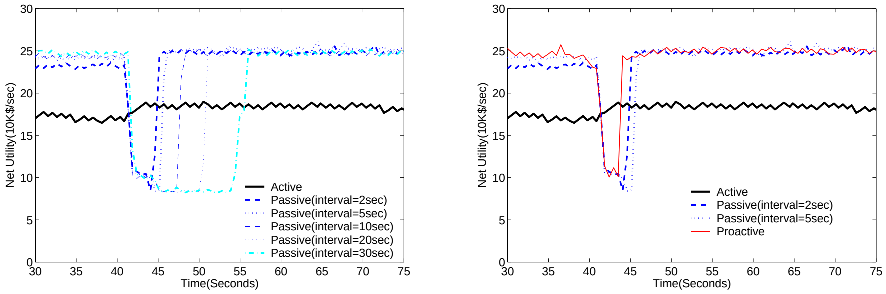
Table 1: Self-Determining Availability based on Benefit

Optimization Criterion	**Utility**	**Cost**	**Delay**
Net-Utility (dollars/sec)	431991	52670	2160
Cost (dollars/sec)	79875	14771	80315
Delay (msec)	222	444	191
f_{cp} (sec^{-1})	0.050	0.018	0.020
Availability (percent)	99.88	99.66	99.70

supplied benefit function. Next, an end-to-end setup is created on Emulab [22], representing an enterprise-scale information flow to compare our approach against the traditional approaches and to study the effect of different soft-checkpoint intervals and proactivity on aspects like MTTR, recovery cost, and net-utility. Results show that proactive availability-management is effective at providing a low-cost failure resilience for information-flow applications while also maximizing the application's net-utility.

5.1 Simulation Study

We conducted a simulation study to compare our utility-based availability management to simple approaches that are not availability-aware. The simulation made use of a 128 node topology generated using GT-ITM internetwork topology generator [35]. We use the formulation of net-utility U_{net} in which the benefit is determined as: $benefit = k_1 \times (k_2 - delay)^2 \times availability \times availableBandwidth/requiredBandwidth$, and the cost is calculated as: $cost = dataRate \times bandwidthCostPerByte$. Random costs are assigned to the network links, expressed in dollars per byte. We substitute ($k_1 = 1.0$, $k_2 = 150.0$) in the benefit formulation for the following simulation [21]. The MTBF was assumed to be 86400sec. and the MTTR was assumed to be 864sec. for a f_{cp} value of 0.01Hz. We first deployed the flow-graph using the net-utility specification from equation 1 as the optimization criteria and the results are shown in Table 1 under the column labeled 'Utility'. The results show a high achieved net-utility with acceptable values for delay, f_{cp} and availability. The second deployment (under 'Cost') focused instead on minimizing the cost, and used $1/cost$ as the optimization criteria.



(a) ***Active and passive approach(various intervals)***

(b) ***Proactive, active, and passive approach***$(interval = 2s, 5s)$

Figure 6: Net utility rate variations using active, passive or proactive fault tolerance approaches. A failure is injected into one operator node at the time $t = 40s$.

The effect of choosing a different criteria is evident in the reduced cost, achieved by allowing a higher delay and a lower availability (resulting from lower f_{cp}). The final deployment uses $1/delay$ to drive the deployment, as a result a reduction in delay is achieved for the flow-graph but at the expense of net-utility and availability.

5.2 Testbed Experiments using IFLOW

This set of experiments is conducted on Emulab [22], and the network topology is again generated using the GT-ITM internetwork topology generator. In many cases, enterprises would hand tune their topology for availability and performance, instead of using an arbitrary topology. For example, an enterprise may explicitly designate a primary and secondary data center. We use an arbitrary topology in our experiments so we can see how well our techniques perform without the benefit of additional hand tuning. Figure 5 shows the testbed used for our experimental evaluations. Background traffic is generated using cmu-scen-gen [25], injected into the testbed using rate-controlled udp connections. For the testbed depicted in Figure 5, background traffic is composed of 900 CBR connections. We use the utility formulation in Equation 19, to better study the net-utility and the cost associated with checkpointing and failures. Required availability is 99.9

5.2.1 Variation of Net-Utility for Different Approaches

The first experiment studies the variation of net-utility with different availability-management approaches in the presence of failures. For simplicity, only one failure is injected into the system. We conduct experiments with the active replication approach, the passive replication approach with varying soft-checkpoint intervals, and the proactive replication approach. Figure 6 clearly demonstrates that the active replication approach provides the lowest net-utility. This is because of the high amount of replicated communication traffic when using this approach. After a failure, the net-utility of the active approach increases slightly; there is less replication traffic, which is a large cost in our utility function, because the failed node no longer sends replicated output updates. The experiment also corroborates the analysis in Section 3.2: a lower soft-checkpoint interval for the passive approach imposes higher communication cost on the system and therefore, results in lower net-utility. Note that if availability were a predominant factor in our net-utility formulation, then a lower soft-checkpoint interval could have resulted in higher net-utility. The cost of soft-checkpoints is almost negligible when the interval is greater than 5 seconds, but its effect is evident for an interval of 2 seconds.

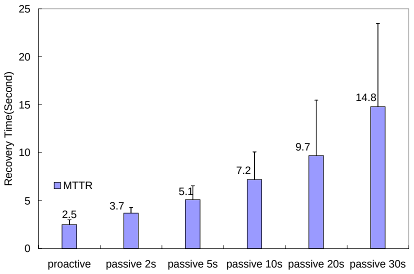


Figure 7: MTTR and standard deviation of recovery time under three replication strategies. Standard deviation is represented by vertical error bars.

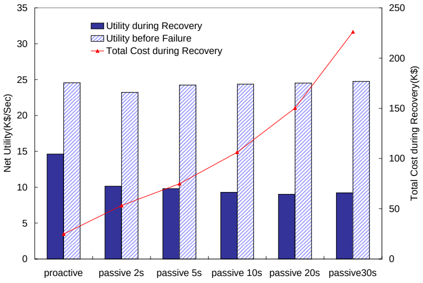


Figure 8: Utility rate and total cost to recover from one failure. For each approach, the left(solid) and right(shadow) bars represent the average utility during failure recovery, and before the failure. The curve with markers represents the total cost including business utility loss during recovery from failure.

The proactive approach provides the highest net-utility overall, as it uses the perceived system stability to modulate the soft-checkpoint interval. For instance, it switches to a smaller soft-checkpoint interval just before the failure and is therefore able to recover as fast as the passive approach with a 2 seconds update interval, while performing as well as the passive approach with a 30 seconds update interval at other times. We note that failure detection is not the focus of this paper. To investigate how prediction accuracy affects the system, these experiments simulate a predictor for the proactive approach, with failure prediction statistically generated at various levels of accuracy. In particular, we notify the soft-checkpoint mechanism that a failure is imminent, no matter whether the prediction is correct or a false positive. In an actual system, faults would be predicted using some learning model (e.g., [10, 34]).

5.2.2 Variation of MTTR for Different Approaches

The variation of MTTR and its standard deviation with different approaches are shown in Figure 7. For each approach, nine experiments are done to obtain the mean and standard deviation. The active replication approach (not shown in the graph) has no explicit recovery time. This is because the node downstream of the replicated operator continues to receive processed updates even after the failure of one active replica. On the other hand, the passive replication approach which attempts to avoid the high cost of active replication incurs recovery times that increase with the soft-checkpoint interval. The reason for this increase is the time taken for reconstructing the operator state: the higher the soft-checkpoint interval, the larger the number of updates required to rebuild the state. Recovery time for the passive replication approach depends on the soft-checkpoint interval. It ranges from 3.7 seconds (for a 2 second interval) to 14.8 seconds (for a 30 second interval). The proactive approach, as expected, performs well as compared to other passive replication approaches, since it is able to change over to a very small soft-checkpoint interval just before the failure, and hence, has low MTTR. The experiment demonstrates the importance of choosing the right soft-checkpoint interval automatically to maximize availability at low cost and thereby maximize the net-utility of information flows.

5.2.3 Cost & Net-Utility During Recovery

The proactive availability-management approach increases soft-checkpoint activity when a failure is predicted in the near future, but it maintains a low soft-checkpoint activity at other times. The analysis of net-utility value before failure, during failure recovery, and the total cost to recover from failure are summarized in Figure 8. The net-utility using proactive availability-management is higher than any other approach, because it contains a very recent soft-checkpoint for the operator state and therefore, incurs the least cost during recovery. Note that passive replication with an interval of 2 seconds also incurs a low cost during recovery, but this is achieved by losing non-negligible net-utility at normal operation time.

5.2.4 Effects of Checkpoint Frequency and Prediction Accuracy on Cost and Availability

The next experiment closely examines the effect of checkpoint frequency on the system, both in terms of system availability and the cost imposed to gain a unit amount of utility. As mentioned in Section 3.2, a higher f_{cp} leads to a higher number of soft-checkpoint messages from the active to the passive node, but it also leads to a smaller number of updates being required to reconstruct the operator state during recovery. The conflicting behavior of incurred cost due to f_{cp} is represented in Figure 9 by the two parabolic curves. Ideally, we would like to spend the minimum cost to achieve a unit amount of utility and would therefore, like to choose a value of f_{cp} that is located at the dip of the parabolic curve. Note that the cost/utility ratio is consistently higher for the passive vs. the proactive approach. We also show the effect of f_{cp} on the availability of the system: the change is in line with the formulation described in Equation 4. However, the interesting insight from this experiment is the direct correspondence between the lowest achievable cost/utility and the flattening of the availability curve.

Our final experiments study the effect of prediction accuracy λ, on the achieved cost/utility ratio. It is intuitive that better prediction accuracy would lead to lower cost/utility for proactive availability-management, and this is clearly depicted in Figure 10. It is interesting to note the behavior of proactive availability-management with a lower f_{cp} value. When prediction accuracy is low, a small f_{cp} leads to very high recovery times with low net-utility during that period. However, if prediction accuracy is high and f_{cp} is modulated to handle such failures, recovery time decreases and a far lower cost/utility is achieved. The effect of prediction accuracy is less prominent when a higher value of f_{cp} is used, as the recovery times don't improve much, even with a correct prediction.

6 Related Work

Traditional Fault-Tolerance. Redundancy is probably the earliest form of fault-tolerance; the approach popularly known as the active replication approach is well-studied, and a thorough description appears in [26]. Log-based recovery is well-know in the database domain. Here, a failure is handled with an undo-redo log [15]. Fault-tolerance has also been studied in the context of transactions [5] and distributed systems [29]. A number of factors distinguish our approach from these traditional mechanism, the first and the foremost being its utility-awareness. Another distinction is our ability to use failure prediction to reduce the overhead of ensuring high-availability.

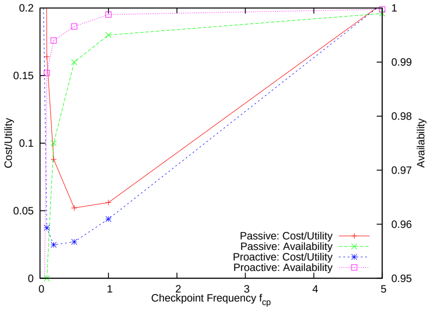


Figure 9: Effect of checkpoint frequency on cost and availability. Checkpoint frequency affects the cost (left y-axis) in a non-linear way, and it is important to optimize it. Note that there is also a sweet spot in the graph, where cost is minimized and availability (right y-axis) is also high. The proactive approach can achieve the same level of availability with significantly less cost compared to the passive approach.

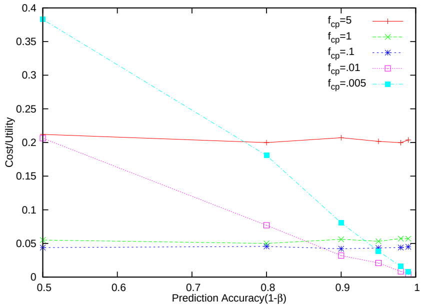


Figure 10: Effect of prediction accuracy on cost of ensuring availability. Better prediction accuracy helps reduce the cost incurred for ensuring high-availability, especially when the checkpoint frequency is high (the curve with the deepest slope). When checkpoint frequency is sufficiently large (the four curves with $f_{cp} \geq 1Hz$), lower accuracy has less effect on the cost due to the fact that less time is required to recover from an unpredicted failure.

Failure Detection & Prediction. [30] and [31] focus on the implementation of fault detection, and [30] proposed a scalable fault detection/collection framework. More recently, researchers in the autonomic domain have used statistical monitoring techniques to detect failures in component-based Internet services [12, 19]. MSET or multi-variate state estimation techniques [34] constitute an early warning system that enables failure prediction with low false alarm probability and has been successfully applied to the thermal control domain, and more recently, to software aging problems, including predicting memory leaks, data corruption, shared memory pool latching, etc. In [10], instrumentation data is correlated to system states using statistical induction techniques to identify system-level metrics that correlate with high-level performance states. In addition, these techniques are used to forecast service level objective violations, with prediction accuracy reported to be around 90%. Failure diagnosis has also been studied in the context of self-managing systems [7]; this could allow developers to embed self-healing features into their systems. Our system provides a framework in which several such failure detection and prediction techniques can be implemented to provide high-availability while imposing a low-overhead.

Fault-Tolerant Distributed Information Systems. Stars [29] presents a fault-tolerance manager for distributed application, using a distributed file manager which performs actions like message backups and checkpoints storage for user files. Its reliance on causal and atomic group multi-cast [27], however, demands additional solutions in the context of today's widely geographically distributed enterprise systems [8].

MTTR may be improved with solutions like Microreboot [4], which proposes a fast recovery technique for large systems. It is based on the observation that a significant fraction of software failures in large-scale Internet systems can be cured by rebooting. While rebooting can be expensive and cause nontrivial service disruption, microrebooting is a fine-grain technique for surgically recovering faulty application components, without disturbing the rest of the components of the application. Our work could benefit from such techniques.

IFLOW's techniques may be directly compared to the fault-tolerance offered in systems like Fault-Tolerant CORBA [17], Arjuna [24] and REL [13], which replicate selected application/service objects. Multiple replicas allow an object to continue to provide service even when one of its replicas fails. Passive replication is also provided. Here, the system records both the state of the currently executing member (primary member) and the entire sequence of method invocations. While CORBA focuses on the client-server model of communication, recent systems like Borealis [2] and SMILE [32] have focused on fault-tolerance for applications that process data streams. The former uses replication-based fault-recovery, and the authors propose to trade consistency for recovery time. The latter proposes the soft-checkpointing mechanism that can be used to implement a low-overhead passive replication scheme for fault tolerance. We differ from such earlier work because of our explicit consideration of system utility for managing system availability, and because our system also provides a framework for incorporating failure prediction techniques.

Utility-Functions. The specific notions of utility used in this paper mirror the work presented in [33], which uses utility functions for autonomic data-centers. Autonomic self-optimization according to business objectives is studied in [1], and self-management of information flow applications in accordance with utility functions is studied in [21]. A preliminary discussion about availability-aware self-configuration in autonomic systems appears in [9]. Our middleware carefully integrates the ideas from the above systems and other domains to build a comprehensive framework for fault-tolerant information flows.

7 Conclusion

We have proposed techniques for managing the tradeoff between availability and cost in information flow middleware. First, a net-utility-based formulation of the benefits an enterprise derives from its information flows combines both performance and reliability attributes of such flows. The goal is not simply to attain high utility, but to reliably provide high utility to large-scale information flow applications. Second, since reliability techniques incur costs thereby reducing utility, proactive methods for availability-management take into account the fact that system and application behaviors change over time. A specific example is a higher likelihood of failure in high load vs. low load conditions. Reliability costs, therefore, are reduced by exploiting knowledge about the current 'perceived' system stability. Additional cost savings result from the use of failure prediction methods. Third, the implementation presented in this paper can deal with both transient and non-transient failures, the latter relying on application-specific techniques for fault avoidance. Finally, utility-driven proactive availability-management technique has been integrated into a representative infrastructure for large-scale information flows, where it is shown to impose low additional communication and processing overheads on information flows. Experimental results with IFLOW attained on Emulab [22] demonstrate the effectiveness of proactive fault tolerance in recovering from failures.

Future work will experiment with richer failure prediction techniques, and it will investigate realistic enterprise environments. We will model the redundant data-centers mandated by government rules, and we will consider the attainment of high availability and net-utility in information flows that cross multiple organizational boundaries.

References

[1] S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, and S. Wasserkrug. Autonomic self-optimization according to business objectives. In *Proc. of ICAC*, 2004.

[2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proc. of the ACM SIGMOD international conference on Management of data*, 2005.

[3] D. Blough and P. Liu. Fimd-mpi: A tool for injecting faults into mpi applications. In *IPDPS*, 2000.

[4] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *Proc. of OSDI*, 2004.

[5] K. Cassidy, K. Gross, and A. Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In *Proc. of DSN*, 2002.

[6] K. J. Cassidy, K. C. Gross, and A. Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. In *Proc. of DSN*, 2002.

[7] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proc. of ICAC*, 2004.

[8] D. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proc. of SOSP*, 1993.

[9] D. M. Chess, V. Kumar, A. Segal, and I. Whalley. Availability-aware self-configuration in autonomic systems. In *Distributed Systems Operations and Management*, 2003.

[10] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. of OSDI*, 2004.

[11] M. J. etc. Comments on mset. http://swig.stanford.edu/ fox/cs444a/miraclecure.html.

[12] A. Fox, E. Kiciman, and D. Patterson. Combining statistical monitoring and predictable recovery for self-management. In *Proc. of the 1st ACM SIGSOFT workshop on Self-managed systems*, 2004.

[13] T. Friese, J. Muller, and B. Freisleben. Self-healing execution of business processes based on a peer-to-peer service architecture. In *Proc. of ICAC*, 2005.

[14] A. Gavrilovska, K. Schwan, and V. Oleson. A practical approach for zero' downtime in an operational information system. In *Proc. of ICDCS*, 2002.

[15] J. Gray, P. R. McJones, M. W. Blasgen, B. G. Lindsay, R. A. Lorie, T. G. Price, G. R. Putzolu, and I. L. Traiger. The recovery manager of the system R database manager. In *ACM Computing Surveys, vol. 13, no. 2*, 1981.

[16] K. C. Gross and W. Lu. Early detection of signal and process anomalies in enterprise computing systems. In *Proc. of IEEE International Conference on Machine Learning and Applications*, 2002.

[17] O. M. Group. Final adopted specification for Fault Tolerant CORBA. In *OMG Technical Committee Document ptc/00-04-04*, 2000.

[18] IBM Global Services. Improving systems availability. http://www.cs.cmu.edu/~priya/hawht.pdf.

[19] E. Kiciman. *Using Statistical Monitoring to Detect Failures in Internet Services*. PhD thesis, Stanford University, 2005.

[20] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-aware distributed stream management using dynamic overlays. In *Proc. of ICDCS*, 2005.

[21] V. Kumar, B. F. Cooper, and K. Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proc. of ICAC*, 2005.

[22] J. Lepreau and et. al. The Utah Network Testbed. http://www.emulab.net/. University of Utah.

[23] M. Mansour and K. Schwan. I-rmi: Performance isolation in information flow applications. In *Proc. of ACM/IFIP/IEEE Middleware*, 2005.

[24] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The design and implementation of arjuna. *USENIX Computing Systems*, 1995.

[25] V. Project. The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[26] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2), 1978.

[27] A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[28] K. Schwan et al. Autoflow: Autonomic information flows for critical information systems. In *Autonomic Computing: Concepts, Infrastructure, and Applications, ed. Manish Parashar and Salim Hariri, CRC Press*, 2006.

[29] P. Sens and B. Folliot. STAR: A fault-tolerant system for distributed applications. *Software - Practice and Experience*, 28(10), 1998.

[30] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. V. Laszewski. A fault detection service for wide area distributed computations. In *Proc. of HPDC*, 1998.

[31] R. Sterritt and S. Chung. Personal autonomic computing self-healing tool. In *Proc. of the IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, 2004.

[32] R. E. Strom. Fault-tolerance in the smile stateful publish-subscribe system. In *Proc. of the Int'l Workshop on Distributed Event-Based Systems*, 2004.

[33] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proc. of ICAC*, 2004.

[34] N. Zavaljevski and K. C. Gross. Uncertainty analysis for multivariate state estimation in mission-critical and safety-critical applications. In *Proc. MARCON*, 2000.

[35] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. of IEEE INFOCOM*, March 1996.