Supporting Fault-Tolerance in Streaming Grid Applications

Qian Zhu  Liang Chen  Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University
Columbus, OH, 43210
{zhuq,chenlia,agrawal}@cse.ohio-state.edu

Abstract—This paper considers the problem of supporting and efficiently implementing fault-tolerance for tightly-coupled and pipelined applications, especially streaming applications, in a grid environment. We provide an alternative to basic checkpointing and use the notion of Light-weight Summary Structure (LSS) to enable efficient failure-recovery. The idea behind LSS is that at certain points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allows us to store copies of LSS for enabling failure-recovery, which causes low overhead fault-tolerance. Our work can be viewed as an optimization and adaptation of the idea of application-level checkpointing to a different execution environment, and for a different class of applications.

Our implementation and evaluation of LSS based failure-recovery has been in the context of the GATES (Grid-based AdapTive Execution on Streams) middleware. An observation we use for providing very low overhead support for fault-tolerance is that algorithms analyzing data streams are only allowed to take a single pass over data, which means they only perform approximate processing. Therefore, we believe that in supporting fault-tolerant execution for these applications, it is acceptable not to analyze a small number of packets of data during failure-recovery. We show how we perform failure-recovery and also demonstrate how we could use additional buffers to limit data loss during the recovery procedure. We also present an efficient algorithm for allocating a new computation resource for failure-recovery at runtime. We have extensively evaluated our implementation using three stream data processing applications, and shown that the use of LSS allows effective and low-overhead failure-recovery.

I. INTRODUCTION

Over the last 10-15 years, an important development in high-end computing has been the increasing use of both commodity and distributed resources for solving large-scale problems. This includes clusters built from off-the-shelf components and use of grid environments, which often comprise such clusters, for solving scientific problems. This development has enabled better cost-effectiveness and flexibility in resource utilization. However, a critical challenge for applications using such environments is that they should be able to deal with failure of individual processing nodes or network links during their execution. As a result, fault-tolerance has become an important topic in high-end computing. For example, within the last 3 years, several projects have developed fault-tolerant MPI implementations1.

Fault-tolerance has recently been studied for grid applications as well. Most of the earlier work on grid computing focused on bag of tasks or the master-worker class of applications. Support for fault-tolerance is relatively easy to provide for such applications, as has been shown by recent efforts [22], [38]. However, in recent years, there has been a growing trend towards supporting more tightly coupled applications. Examples of such application classes include scientific workflows [1], [13], applications that use pipelined or data-flow like processing [5], and streaming applications [10], [33].

Fault-tolerance is even more important for these applications, because of two major reasons. The first is the long-running nature of these applications. The second is that these applications often require frequent and large volume data transfers between processing stages, and network failures can also disrupt execution of these applications. At the same time, fault-tolerance is harder to achieve for these applications, because significant amount of state can be associated with each processing stage, and allocation of resources for each stage cannot be done independently.

This paper considers the problem of efficiently implementing fault-tolerance for tightly-coupled and pipelined applications, especially streaming applications, in a grid environment. Most streaming applications require results to be continually produced at a low latency, even in the face of highly variable data input rates. In this paper, we focus on the stop-and-fail fault. Our fault-tolerance framework is able to deliver correct results in a timely manner in the presence of faults, i.e., it appropriately balances the trade-off between consistency and availability. Previous works favor using replication to mask failures, by requiring at least one complete processing chain to exist to continue execution at any time [22]. In contrast, the proposed scheme eliminates the overhead of running multiple copies of the same work on distinct processing nodes. To be specific, we provide an alternative to basic checkpointing [31]. We use the notion of Light-weight Summary Structure (LSS) to enable failure-recovery. The idea behind LSS is that at certain

1See, as examples, icl.cs.utk.edu/ftmpi and www.linux-mag.com/2004-11/fault_01.html
points during the execution of a processing stage, the state of the program can be summarized by a small amount of memory. This allows us to store copies of LSS for enabling failure-recovery, which allows low overhead fault-tolerance.

Our LSS-based approach is similar to the application-level checkpointing for parallel programs [17], [18], [16]. The work by Bronevetsky et al. has investigated the use of compiler technology to instrument codes to enable self-checkpointing and self-restarting, thereby providing an automatic solution to the problem of making long-running scientific applications resilient to hardware faults. Our work can be viewed as an optimization and adaptation of this work to a different execution environment, and for a specific class of applications. In particular, LSS considers the specific characteristics of streaming applications and provides efficiency in fault-tolerance. Since LSS captures the essential information in a compact way for the execution of streaming applications, failure-recovery is fast, while maintaining a high consistency in results. Moreover, due to the small size of LSS, we can significantly reduce the overhead of transferring LSS for storage.

Our implementation and evaluation of LSS-based failure-recovery have been in the context of the GATES (Grid-based AdapTive Execution on Streams) middleware [10], [9]. GATES system has been designed to support processing of distributed data streams in a wide-area environment. In the stream processing model, data arrives continuously and needs to be processed in real-time, i.e., the processing rate must match the arrival rate. An observation we use for providing very low overhead support for fault-tolerance is that algorithms analyzing data streams are only allowed to take a single pass over data, which means they only perform approximate processing. Therefore, we believe that in supporting fault-tolerant execution for these applications, it is acceptable to not analyze a small number of packets of data during failure-recovery.

The overall contributions of this paper are as follows. We have demonstrated an implementation of fault-tolerance using LSS in the context of the GATES middleware. We show how we perform failure-recovery. Furthermore, we describe how we could use additional buffers to limit data loss during failure-recovery. We also present an efficient algorithm for allocating a new computation resource for failure-recovery at runtime.

We have extensively evaluated our implementation using three data streaming applications. The main observations from our experiments are as follows. Firstly, the size of LSS is almost two orders of magnitude smaller than the total memory required by the application and the middleware, which points to the efficiency of our approach. Secondly, the overhead of storing and copying LSS our middleware is almost negligible. The cost associated with failure-recovery is also modest. This overhead increases somewhat in the version where additional buffering is done to limit data loss or when the node for failure-recovery has to be determined at runtime. Similarly, the loss of accuracy is very small, and is further reduced when data loss is limited. However, it increases if the node for failure-recovery needs to be identified at runtime. Finally, we have shown that our method for choosing a new computation resource at runtime is effective, and failure-recovery at such nodes gives better performance than other possible options.

II. DISTRIBUTED STREAMING APPLICATIONS AND GATES Middleware

This section briefly summarizes the distributed streaming class of applications and lists the major design aspects of the GATES system.

Increasingly, a number of applications across computer sciences and other science and engineering disciplines rely on, or can potentially benefit from, analysis and monitoring of data streams. In the stream model of processing, data arrives continuously and needs to be processed in real-time, i.e., the processing rate must match the arrival rate. There are two trends contributing to the emergence of this model. First, scientific simulations and increasing numbers of high-precision data collection instruments (e.g. sensors attached to satellites and medical imaging modalities) are generating data continuously, and at a high rate. The second is the rapid improvements in the technologies for Wide Area Networking (WAN), as evidenced, for example, by the National Lambda Rail (NLR) effort and the interconnectivity between the Tera-Grid and Extensible Terascale Facility (ETF) sites. As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster.

The important characteristics that apply across a number of stream-based applications are: 1) the data arrives continuously, 24 hours a day and 7 days a week, 2) the volume of data is enormous, typically tens or hundreds of gigabytes a day, and the desired analysis could also require large computations, 3) often, this data arrives at a distributed set of locations, and all data cannot be communicated to a single site, 4) it is often not feasible to store all data for processing at a later time, thereby, requiring analysis in real-time.

We briefly describe two representative examples. The first application we consider is online network intrusion detection, which is a critical step for cyber-security. Online analysis of streams of connection request logs and identifying unusual patterns is considered useful for network intrusion detection [14]. To be really effective, it is desirable that this analysis of logs at a number of sites can be performed in a distributed fashion. Large volumes of data and the need for real-time response make such analysis challenging. The second example is computer vision based surveillance. Multiple cameras shooting images from different perspectives can capture more information about a scene or a set of scenes. This can enable tracking of people and monitoring of a critical infrastructure [6]. A recent report indicated that real-time analysis of the capture of more than three digital cameras is not possible on current desktops, as the typical analysis requires large computations. Distributed and grid-based processing can enable such analysis, especially when the cameras are physically distributed and/or high bandwidth networking is available.
The distributed streaming processing applications we consider are somewhat related to the content distribution applications, which have received much attention lately [21], [24], [25], [26]. However, there is a very important distinction between these two classes of applications. Content distribution applications simply deliver a data stream to a set of hosts. In comparison, the applications we consider perform non-trivial computations, in a pipelined fashion over a series of processing nodes.

The problem of flexible and adaptive processing of distributed data streams can be viewed as a grid computing problem. A distributed and networked collection of computing resources can be used for analysis or processing of these data streams. Computing resources close to the source of a data stream can be used for initial processing of the data stream, thereby reducing the volume of data that needs to be communicated. Other computing resources can be used for more expensive and/or centralized processing of data from all sources. Because of the real-time requirements, there is a need for adapting the processing in such a distributed environment, and achieving the best accuracy of the results within the real-time constraint.

There are three main goals behind the design of the GATES system. The first is to enable the application to achieve the best accuracy, while maintaining the real-time constraint. For this, the middleware allows the application developers to expose one or more adaptation parameters at each stage. The middleware automatically adjusts the values of these parameters to meet the real-time constraint on processing. This is achieved through a self-adaptation algorithm [10], [9]. The second is to support distributed processing of one or more data streams, by facilitating applications that comprise a set of stages. All intermediate stages take one or more intermediate streams as input and produce one or more output streams. GATES’s API is designed to facilitate specification of such stages. Finally, the last goal is to enable easy deployment of the application. This is done by supporting a Launcher and a Deployer. The system is responsible for initiating the different stages of the computation at different resources. The system also allows the use of existing grid infrastructure.

III. OVERALL DESIGN FOR FAULT-TOLERANCE

This section describes our approach for supporting fault-tolerance in distributed streaming applications.

A. Design Alternatives

As a variety of commodity resources are used in a grid environment, unexpected failures of resources are quite likely. Therefore, tolerating faults in order to complete applications becomes an important issue while developing, deploying, and executing applications and associated middleware. In particular, long running applications, such as applications that process streaming data over a long period of time, critically need to continue with the execution even if one of the processing nodes fails. A grid middleware supporting such applications should allow the application to produce reasonably accurate results with a tolerable delay in the presence of failures.

Classically, fault-tolerance has been supported by using redundancy to execute multiple copies or replicas of jobs concurrently. We are specifically referring to replication in space, which is the additional use of resources. The purpose of using replicas is to continue the processing by switching to a replica in the case of failures of the node executing the primary version of the job. It is obvious that with replication, the system can maintain high availability even when faults occur. However, we see the following three issues with this approach. First, while having $m$ replicas can help tolerate up to $m-1$ simultaneous failures, it also involves a factor $m$ increase in computational requirements. Second, all replicas need to maintain identical processing states and in resuming execution, we need to retrieve the exact failure point while switching to another replica. This poses a challenge in performing synchronization of states across the replicas. Finally, network connectivity and communication bandwidth requirements also increase with increasing number of replicas.

Besides replication, checkpointing is also a well-known technique for reducing the loss of computational progress when a failure occurs. Checkpoint-based fault-tolerance involves taking a periodic snapshot of the system’s running state, which includes processes’s execution points, memory stacks (pages), and the CPU status. This state is saved on a new node, where the original execution environment is restored and processes can restart at the point where the checkpoint was taken.

Using process checkpointing for fault-tolerance in a grid environment poses several challenges. First, checkpoints are usually platform dependent, whereas, a grid comprises of heterogeneous resources, and grid standards have been designed to support applications independent of hardware and operating systems. Thus, using basic checkpointing to tolerate faults is not practical in a grid environment. Furthermore, large-volume checkpoints can result in inefficient failure-recovery, especially for data streaming applications. These applications process a large volume of data in memory. Because a checkpoint includes memory images, this can imply significant overhead in taking and transmitting such large-volume checkpoints. The impact can be even more severe for data streaming applications, because they need to meet real-time constraint on processing data.

B. Our Approach

We now describe our approach for supporting failure-recovery, which is based on the notion of a Light-Weight Summary Structure (LSS).

The design of LSS is based on the observation that for many application classes, including data stream processing and other pipeline/data-flow like systems [5], the processing structure is as follows:

```java
while(true) {
  ...
}
```
read_data_from_streams();
process_data();
accumulate_intermediate_results();
}

During each loop, a number of data items from the stream are read and processed. The processing results are accumulated to form a summary information. We name the data structure storing such summary information as the Light-Weight Summary Structure (LSS). Other data structures used by the application are considered Auxiliary Structures. Memory locations in auxiliary structures are always reset to initial values at the end of each loop.

Two additional observations are important with respect to LSS. First, for most applications, the size of LSS is much smaller than that of auxiliary structures, and therefore, also much smaller than the total memory used by the application. Second, since auxiliary structures are anyways reset at the end of each loop iteration, they do not contribute to the execution state, if the state is recorded at the end of a loop iteration.

Another observation with respect to use of LSS and supporting fault-tolerance for our target class of applications is as follows. Algorithms analyzing data streams are only allowed to take a single pass over the data. Therefore, these algorithms perform only approximate processing. Often, to maintain real-time constraint in processing, data is sampled. Therefore, we believe that in supporting fault-tolerant execution for these applications, it is acceptable to not analyze a small number of packets of data during failure-recovery.

Based on these observations, LSS can be used for fault-tolerance, specifically, efficient failure-recovery. The middleware can provide a function to periodically store LSS corresponding to a particular stage at a remote location. Once the service on this stage fails or the node accommodating it crashes, the middleware restarts the service on the remote location and retrieves locally stored LSS for a fast recovery. The application then resumes from the latest stable execution point, possibly incurring some loss of accuracy in processing.

Failure-recovery supported by LSS can be more efficient and suitable to grid environments compared to using normal checkpointing in the following ways:

- As noted earlier, only LSS, a much smaller portion of the overall memory needed by the applications, is copied and stored.
- Despite achieving efficiency, we do not significantly impact the accuracy of processing. Though data is lost during failure-recovery, we believe that this can be tolerated in data streaming applications because algorithms are only approximate in nature.
- LSS is a logical data-structure and its contents are quite independent of the specific platform. We do not need program contexts to migrate, and therefore, it is easier to support fault-tolerance across heterogeneous platforms.

As we noted earlier, our LSS based approach has many similarities with the recent work on application-level checkpointing [17], [18], [16]. The key difference is that we focus on a specialized class of applications, and rely on application developers to identify a small subset of application’s memory space that captures the state at the end of each processing round. Thus, our can be viewed as an optimization and adaptation of application-level checkpointing to a different execution environment, and for a specific class of applications.

C. Middleware Implementation

We now present details of our implementation for fault-tolerance in GATES middleware. The pseudo-code in Figure 1 shows how an application can cooperate with GATES to remotely store LSS. Each application needs to implement its own LSS class that declares the summary structure as its member and decides the application data to be stored in LSS. In the application example shown here, the LSS class is counting-lss.

We can use getLSS(Name-of-LSS-Class) to specify a LSS class to GATES. GATES will return an instance of the LSS class, which is either created locally, or is retrieved from local LSS at the remote site, in the case processing restarts on that node. We separate the strategy of storing LSS from the execution of the application. Transmitting LSS remotely could lower system performance in the situation where a service frequently updates its LSS, and network bandwidth is low between the node where the service resides on and the remote location for storing LSS. To achieve a balance between system efficiency and availability, we use caching mechanism to solve this problem. Specifically, the middleware allocates a memory block on the node for a local copy of LSS and lets the service update it. As processing data is more time-consuming than replicating LSS, a replication service can transmit local LSS for remote storage during data processing. Thus, by overlapping these two operations, the system can

---

Fig. 1. Pseudo-code for an Application using GATES Fault-Tolerance API
achieve high availability, with minimal impact on performance, as we will show later through our experimental results.

We used a heartbeat-based method for fault-detection. When a fault is detected, GATES will set the status of this input buffer to be invalid. Thus, at the beginning of each loop, the input buffer status is checked to see whether it has been invalidated. If so, all processing on this node stops and the newly selected node will continue with the work. GATES implementation of the failure-recovery procedure is explained in Figure 2.

LSS is locally updated at the end of each application processing loop. Once GATES is notified, the replication service will transmit LSS to all the candidate nodes, which are chosen for LSS storage. This operation overlaps with data processing. Meanwhile, each candidate updates its memory block for LSS correspondingly. A new node with LSS copy and high network bandwidth with other nodes should be selected right after a fault has been detected. This node can be pre-decided by the user, or chosen from the resource location algorithm, as will be discussed in Section IV-B. As shown in the Figure 2, N2 is the optimal replacement for node B. The reason is that, first, the latest version of LSS is stored on N2. Second, N2 connects to both A and C with high bandwidth. Thus, we can see that switching to N2 will result in high rate of execution for the continuing application.

The procedure for failure-recovery starts by creating a new path from A to N2 to C. This includes currently establishing socket connections between A and N2, and between N2 and C, and initializing internal buffers for these socket connections. Next, GATES stops sending data from A to B. At this time, there could be data left in the obsolete input buffers in A, also potentially in B, if fault-detection was incorrect or if B comes back up. We simply drop or ignore this data, since it allows simplified implementation, and, for our target class of applications, acceptable accuracy. In Section IV-A, we will describe a technique for limiting the loss of accuracy. At the third step, A’s output buffer and C’s input buffer are set to be invalid at the same time. Thus, A will forward data to N2 instead of B, and similarly, C will receive data only from N2 now. Since LSS is stored on N2, the fourth step is loading the corresponding application code into N2 and initiating its execution. All the fault-tolerance functionality can be used for multiple stages within an application, and will also apply to the new configuration in order to deal with any further failures.

IV. OTHER ISSUES AND ENHANCEMENTS

This section focuses on two other issues and enhancements to our scheme. The first issue we address is as follows. During the process of failure-recovery, data can be lost due to the time elapsed while locating a new node for the processing path, as we discussed previously. Losing too much data significantly impacts the accuracy. Thus, we propose a new strategy, i.e., using a data backup buffer. Second, for the cases when the computing node for failure-recovery cannot be selected in advance, we present an algorithm for fast allocation of a new processing node. This algorithm has been designed with two goals. First, it needs to choose a node which will allow efficient execution in the future. Second, the algorithm should be fast, since we need to minimize the time spend (and therefore, the data loss) during failure-recovery.

A. Data Backup Buffer

Data transferred from the upstream to the downstream is in the unit of a packet, which holds a constant number of data points. We assign a sequence number or timestamp to each data packet as its unique identity.

The key idea in our approach is as follows. On the upstream node, each time a data packet is sent out, we also place it into a backup data buffer. This packet is kept till either an acknowledgment has been received, or, it has to be replaced by a new packet. A consideration here is the buffer size, which cannot increase indefinitely. If the downstream node sends out an acknowledgment upon receiving every m data packets, we are willing to allow at most k packets to be lost, and we allow a timeout of t packets, we need a buffer of sufficient size to hold m + t - k packets. This is because we need to process
m + t packets before we expect an acknowledgment, but we can remove up to k packets from the buffer.

On the downstream node, each time after receiving m data packets from its upstream neighbor, an acknowledgment with the largest sequence number of packets, say n is sent. By doing this, we confirm that it has received packets n − m + 1, . . . , n. On the arrival of such an acknowledgment, the upstream node simply drops m data packets from the backup buffer with sequence number less than or equal to n.

In the case of failure on the downstream node, no acknowledgment will be sent back to the upstream node. Suppose the upstream node is ready to output packet with timestamp N to the buffer. If an acknowledgment with timestamp at least N + k − m − t has not been received, it will remove the oldest packet from the buffer, and add the packet with the timestamp N to the buffer. Now, let us consider the situation when it is ready to output the packet with timestamp N + k to the buffer. If it still has not received an acknowledgment with timestamp N + k − m − t, it cannot remove any further packets from the buffer. In this case, it must conclude that a failure has occurred, and needs to start failure-recovery. All the packets currently in the buffer will now be processed at the new node.

This mechanism ensures that the maximum number of packets that may not be processed during a failure will be k. Clearly, there is a trade-off between the parameters. A larger value of k reduces the size of the buffer, but can increase the level of inaccuracy of the final results. A larger value of t avoids false failure discovery, but increases buffer size. Finally, a larger value of m increases the buffer size, but reduces the overhead associated with acknowledgment messages. For simplicity, our current implementation assumes m = t = k, which means that a buffer of size m is required.

B. Efficient Resource Allocation Algorithm

The data loss can also be decreased by fast allocation of the new processing node without any backup buffer for temporary data storage. The algorithm we propose for efficient resource allocation works as follows. We choose a certain number of computing nodes as a candidate set and finally pick one from this set as the new processing node. Intuitively, we have the following two requirements:

- Each node in the candidate set should have high network bandwidth connection to the node which generates LSS, denoted as B. This ensures that the remote storage of LSS can be done in a timely manner and with limited overhead.
- The new processing node chosen from the candidate set should be connected to both its upstream node, denoted as A, and downstream node, denoted as C, with high network bandwidth. Thus, we can maintain high execution rate even in the case of failure of node B.

We assume that the topology of the network is a connected graph, with each computing node as a vertex, the connection between two nodes as an edge and the reciprocal of the bandwidth as the weight of each edge. Hence, a small edge weight denotes a high bandwidth between the two computing nodes. The network latency is determined by using the network weather service [35] and we assume that the bandwidth information is known to GATES. In order to locate the nodes connected to B by high-bandwidth network, the algorithm starts with running the Dijkstra’s shortest path [15] algorithm on B. Note that P2P overlay topologies [30] could also be used here. We set a user-defined threshold and any path shorter than this value is considered as a high-bandwidth connection. Simultaneously, we also run the Dijkstra’s shortest path algorithm on nodes A and C, in order to locate the nodes which are connected to each of them with high bandwidth.

The candidate set comprises nodes from the results of running Dijkstra’s shortest path algorithm on A, B, and C. We assign a priority to each node in the set as follows, with 3 as the highest priority.

- Assign 3 to the nodes which are the intersection of three result sets. These nodes have high bandwidth connections to all A, B, and C. According to the requirements, these are the best candidates.
- Assign 2 to the nodes which are the intersection of result sets from A and B. As more data can be transferred from A to B than those from B to C, high bandwidth is more important.
- Assign 1 to the nodes which are the intersection of result sets from B and C.
- Assign 0 to the nodes which are the intersection of result sets from A and C. These nodes satisfy the second requirement and would be best candidates once LSS is stored on them.

The network topology is changing dynamically and unpredictably, and so is the graph. Thus, we have to update the candidate set, as well as the priority assigned to each node of the set, in order to satisfy the requirements above. For the nodes with priority 3, we also associate them with a level value. The level value increases as the node remains in the candidate set. The reason is that we believe this high bandwidth connection is more stable if the node can stay in the set for a longer time. We update the nodes using the rules for priority assignment, based on variations of network connections. Furthermore, once LSS can be stored on the nodes with priority 0, we change it to be 3 since they satisfy the two requirements and become the best candidates then. To choose the best candidate for the new processing node, we pick the one with the highest priority and level value, if possible.

We update the candidate set after a time interval by running Dijkstra’s shortest path algorithm on node A, B and C. The environment node (Enode) is responsible for monitoring the network topology and choosing nodes for the set. Then, at the end of each processing round, node B would inquire Enode for the locations to remotely store LSS. Also, when there is a need for a new node, Enode selects the best candidate from the set and sends it to node A for creating a new path.

The algorithm we proposed can locate the new processing node quickly before the creation of the new path. This also results in reduced data loss, since less time is elapsed during
the procedure of choosing the new node. Furthermore, as we will show through our experiments, the node picked by the algorithm results in lower execution time than the other choices.

V. Streaming Applications

This section describes three applications that were used to evaluate our scheme for supporting failure-recovery.

a) Counting Samples:: count-samps is a distributed version of the counting samples problem. The classical counting samples problem is as follows. A data stream comprises a set of integers. We are interested in determining the \( n \) most frequently occurring values and their number of occurrences at any given point in the stream. Since it is not possible to store all values, a summary structure must be maintained to determine the frequently occurring values. Gibbons and Matias [19] have developed an approximate method for answering such queries with limited memory. The problem we consider is of determining frequently occurring values from distributed streams. Our solution is to store \( m \) frequently occurring values from each stream at a node close to the data source, and then merge them to determine the overall \( n \) most frequently occurring values at a central location. The value of \( m \) can be chosen to provide a trade-off between the accuracy of the final results and the efficiency of processing. These \( m \) values form the LSS at each remote node.

b) Clustering Evolving Streams:: The second application is clustering evolving data streams [2], and is referred to as CluStream. Clustering involves grouping similar objects or data points from a given set into clusters. The particular problem considered here is clustering data arriving in continuous streams, especially as the distribution of data can change over time. The algorithm we consider [2] approaches the problem as follows. The clustering process is divided into two major steps. The first steps involves computing micro-clusters that summarize statistical information in a data stream. In fact, the set of micro-clusters is considered the summary information of this application. The second step uses micro-clusters to compute the final clusters.

This two-step clustering algorithm can be easily implemented using the GATES middleware. We have used three stages for stream processing. The first stage is simply the data source, which sends streaming data to the second stage. The second stage computes micro-clusters. After a certain number of data points have been processed, it sends the computed micro-clusters to the third stage. The third and the final stage then applies the modified k-means algorithm [2] to create and output the final clusters. To make the second stage capable of failure-recovery, we store micro-clusters as an LSS object.

c) Distributed Frequency Counting:: The third application we studied finds frequently occurring itemsets in a distributed data stream and is referred to as Dist-Freq-Counting [27]. The problem is of finding frequently occurring itemsets across a set of data streams. If the distribution of data across the different streams is different, and if the communication bandwidth is limited, this problem can be quite challenging. The algorithm we consider is an extension of the algorithm for finding frequent items from distributed streams proposed by Manjhi et al. [27]. The algorithm addresses the problem stated above by arranging the nodes in a hierarchical structure. Each monitor node \( M_i \) receives itemsets from the data source \( S_i \) and inserts them to a list. After a certain number of itemsets are received, \( M_i \) counts the frequencies of the itemsets appearing in the stream, sends this information to an intermediate node, purges all itemsets in the list and starts a new round. Intermediate nodes combine the frequency information received from their children and pass them up to their parent node. To reduce the communication load, the monitor and intermediate nodes should avoid sending less frequent itemsets over the links. Therefore, the algorithm uses an error tolerance parameter \( \epsilon \) at every node, except the data sources. Only the itemsets with frequencies greater than \( \epsilon \) are forwarded to the next node. Finally, the root node outputs the itemsets whose frequencies exceed the specified support threshold \( \tau \).

To make monitors capable of failure-recovery, we have implemented a LSS class to store the unprocessed itemsets. This is based on the following observation: a failure could happen when the monitor is still accumulating itemsets. These accumulated itemsets, which have not yet been processed, need to be transmitted to the remote node. Therefore, we choose these itemsets as the summary information.

VI. Experimental Results

This section presents results from a number of experiments we conducted to evaluate our approach and implementation. Specifically, we had the following goals in our experiments.

1) Show that LSS uses a small amount of memory for our target applications, compared with the total memory usage of the applications and the middleware. Thus, we show that our approach can be very effective in bandwidth-constrained environments, unlike checkpointing.

2) Show that LSS can be used for fault-tolerance with very little overhead, and without impacting the overall application performance.

3) Show that the accuracy is not significantly decreased in case of a failure using our approach, and even this loss of accuracy can be reduced further if amount of data lost can be controlled.

4) Show that the node chosen by our resource allocation algorithm yields better performance than other possible choices.

A. Experiment Setup and Datasets

For distributed processing of streaming data in a grid environment, we need high bandwidth networks. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single Linux cluster. The cluster consists of 64 computing nodes. Each node has 2 Opteron 250 (2.4GHz) processors with 8GB
of main memory and 500GB local disk space, interconnected with switched 1 Gb/s Ethernet.

We believe that our choice of experimental setup has a very minor impact on the four goals we are focusing on, as listed above. The first and third metrics completely independent of the platform. While the overhead of the approach (the second metric) can increase in a wide-area environment, such overheads will be much higher for other possible approaches, such as those based on replication or checkpointing. For the fourth goal, we are simulating different inter-node bandwidths in a controlled environment.

The experiments were conducted using the three streaming data mining applications described earlier in this paper. For the count-samps application, integer streams were generated by a simulator. For the CluStream application, we used the KDD-CUP’99 Network Intrusion Detection dataset. For Dist-Freq-Counting, we used a dataset generated by the IBM synthetic data generator [3]. We conducted four sets of experiments which we describe in the rest of this section. The first, second and fourth were conducted using all three applications, and only count-samps and CluStream were used for the third experiment, since accuracy was hard to quantify for the Dist-Freq-Counting application.

B. Memory Usage with LSS

This experiment shows that compared to the entire application and the middleware, LSS uses a small fraction of memory.

For count-samps, LSS records the $m$ most frequently occurring numbers and their corresponding frequencies on an intermediate stage. Therefore, we varied the value of $m$ and measured memory usage of LSS and the entire application. The results are shown in Figure 3. When $m$ was set to 20, 80, 160 and 200, LSS only occupied approximately 0.6%, 1.7%, 2.5% and 2.9%, respectively, of memory used by the entire application.

We further examined the memory usage for count-samps, when $m$ was set to 200. The memory use by the middleware, the application, and LSS are 4,350KB, 1,662KB and 48KB, respectively. Thus, LSS just used approximately 0.8% of the total memory consumed by GATES and the application. This clearly points to the efficiency of storing LSS at remote sites for failure-recovery.

<table>
<thead>
<tr>
<th>Value of $m$</th>
<th>Size of LSS (KB)</th>
<th>Size of count-samps (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>6</td>
<td>1,149</td>
</tr>
<tr>
<td>80</td>
<td>20</td>
<td>1,143</td>
</tr>
<tr>
<td>160</td>
<td>36</td>
<td>1,432</td>
</tr>
<tr>
<td>200</td>
<td>48</td>
<td>1,662</td>
</tr>
</tbody>
</table>

Fig. 3. Memory Usage of LSS for count-samps

We repeated the above experiment using CluStream and Dist-Freq-Counting. In the case of CluStream, its LSS is the set of micro-clusters, the size of which depends on the number of micro-clusters. Therefore, we varied the number of micro-clusters and then measured memory usage of LSS and the entire application. The results are shown in Figure 4. Also, when the number of micro-cluster is 100, we measured that LSS only consumed approximately 0.9% of the total memory used by GATES and the application. LSS of Dist-Freq-Counting is the set of unprocessed transactions, and its size is proportional to the number of such transactions. This, in turn, depends on when the fault occurs. Therefore, we restarted the application at six random time instances, and measured the LSS’s memory usage and the corresponding number of unprocessed transactions. The results are indicated in Figure 5. We also measured the average size of Dist-Freq-Counting, which is 16,442KB. Thus, LSS only used on average 1.1% of the total memory consumed by the middleware and the application.

Overall, this set of experiments establish that our approach requires much less memory and bandwidth than the normal checkpointing approach, and is therefore well-suited for a grid environment.

C. Using LSS for Fault-Tolerance: Performance

We compared four different executions for each of the applications to show that our approach for supporting fault-tolerance has low overhead.

The first execution involves a version of the application that cannot support failure-recovery. This version does not use LSS, and is referred to as the Without Fault version. The second and third executions involve versions that are capable of failure-recovery, and during the execution, they incur a

<table>
<thead>
<tr>
<th>Number of Micro-Clusters</th>
<th>Size of LSS (KB)</th>
<th>Size of Dist-Freq-Counting (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
<td>1,067</td>
</tr>
<tr>
<td>20</td>
<td>13</td>
<td>1,143</td>
</tr>
<tr>
<td>40</td>
<td>25</td>
<td>1,437</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>1,673</td>
</tr>
<tr>
<td>100</td>
<td>62</td>
<td>2,382</td>
</tr>
</tbody>
</table>

Fig. 4. Memory Usage of LSS for CluStream

<table>
<thead>
<tr>
<th>Time Period</th>
<th>Number of Transactions</th>
<th>Size of LSS (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>802</td>
<td>177</td>
</tr>
<tr>
<td>2</td>
<td>1557</td>
<td>305</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>226</td>
</tr>
<tr>
<td>4</td>
<td>728</td>
<td>159</td>
</tr>
<tr>
<td>5</td>
<td>2437</td>
<td>464</td>
</tr>
<tr>
<td>6</td>
<td>740</td>
<td>161</td>
</tr>
<tr>
<td>Average</td>
<td>1227</td>
<td>238</td>
</tr>
</tbody>
</table>

Fig. 5. Memory Usage of the LSS for Dist-Freq-Counting

Fig. 6. Execution Time of Four Versions: count-samps
allocation algorithm to choose the new processing node at that both these cases involve a pre-chosen node for failure-recovery. For the fourth execution, we used our resource runtime, and it is referred to as of packets that can be lost (i.e. the parameter \(m\)) is set at 2. The second and third executions are referred to as With Fault & Chosen Node and With Fault & Chosen Node (Losing 2 Packets) versions, respectively. Note that both these cases involve a pre-chosen node for failure-recovery. For the fourth execution, we used our resource allocation algorithm to choose the new processing node at runtime, and it is referred to as With Fault & Runtime Node version. In all the experiments involving second, third, or fourth version, the failure occurred exactly once.

First, we measured the time count-samps takes under the four variations described above, using different number of packets. Each packet comprised 1000 integers. The results are indicated in Figure 6. Due to the overheads of saving and copying LSS, and failure-recovery, the version with fault is slower than the version without fault, even though we restarted the processing on a pre-chosen node. However, the difference between the two versions is small, close to 10% with 1000 packets, and close to 4% with 8000 packets. This shows that our implementation for storing and copying LSS, and failure-recovery is efficient. Because the additional time taken by the second version remains constant as we increase the dataset size, we can see that almost all of the overhead is because of failure-recovery. In other words, we are able to effectively overlap processes for storing and copying LSS with normal processing.

Furthermore, the version where we controlled data loss is slightly slower than the version where we did not. This is due to the overhead of resending data from the backup buffer to the new node. In the case of the version choosing node at runtime, the execution time increases. The reason is that the entire failure-recovery procedure has to wait for the best candidate generated by the algorithm, before it can create a new processing path. However, the net difference between the first version and the fourth version is less than 7% when 8000 packets are sent. Overall, these results show that if a failure occurs rarely, failure-recovery can be supported with very little overhead.

Using Clustream, we varied dataset sizes to 200KB, 400KB, 800KB, 1600KB and 3200KB and compared the execution time of these four versions. Similarly, we repeated the above experiments using Dist-Freq-Counting. We obtained very similar results from both applications. They are presented in Figure 7 and Figure 8, respectively. In the case of Clustream, with 3200 KB dataset, the difference between the first and the fourth version is only 2.5%. In the case of Dist-Freq-Counting, with a 12 MB dataset, the difference between the first and the fourth version is only 3.5%. These results show that allocation of resources, and failure-recovery can be performed efficiently using our approach, especially when a large dataset is processed, and failures are infrequent.

D. Using LSS for Fault-Tolerance: Accuracy

In this subsection, we show that how the fault-tolerance technique using LSS can impact accuracy of data stream processing. As we had stated earlier, we used count-samps and Clustream to conduct this experiment.

For the count-samps application, we synthetically generated an integer stream in which 50% numbers are 1, 25% numbers are 2, 12.5% numbers are 3 and so forth. Thus, the top 10 frequently occurring numbers in the stream are 1,2,3,...,10. We used a criterion considering how many numbers from 1,2,...,10 are picked to measure the accuracy. The highest accuracy is 10, corresponding to the case when each of the numbers 1 to 10 is picked, while the lowest is 0, if none of the numbers from 1 to 10 is picked. We conducted 5 rounds of experiments for each of the four variations. We calculated average accuracy of 5 results, and the results are shown in Figure 9.

Obviously, the accuracy gets worse when a fault occurs during the processing and data loss was not controlled. However, when we used the data backup buffer to control data loss, that is, only 2 packets (or 2000 integers) were lost during failure-recovery, there was an improvement in the accuracy. In the case of the version where we choose the new node at runtime, since more time was consumed for the algorithm to pick the best candidate, there was significantly higher data loss also, which leads to worse accuracy. When the number of packets
is 8000, the loss of accuracy in the results from the fourth version is about 6%, over the first version. Considering that the original one-pass algorithm also had less than 90% accuracy, we believe that such loss of accuracy may be acceptable to most users. If the node for failure-recovery can be identified in advance, and if additional buffers are maintained, the loss of accuracy is less than 1%.

We repeated the same experiment with the Clustream application. To compute the accuracy of this algorithm, we used the sum of square distance (SSD) to measure the quality of clusters. The smaller this value is, the smaller intra-distance is for each cluster, which means better quality clusters. The results are shown in Figures 10. Overall, the results are very similar to those from the previous application, showing that loss of accuracy is very modest.

E. Evaluating Resource Allocation Algorithm

We now investigate whether the efficient resource allocation algorithm chooses a new processing node to give the best overall performance. This experiment involved 32 computing nodes in total. Except the nodes used to accommodate the initial processing stages, all the others can be chosen as the new node. The bandwidth between these nodes was chosen randomly. By adding a delay in message passing, such a heterogeneous configuration was simulated within a single cluster.

Our first experiment used count-samps application. We fixed number of packets to be 8000. In order to check whether this node gives the best performance or not, we also continued the processing using other nodes inside and outside of the candidate set. Recall that nodes in the candidate set have different priority values. We used dot symbol to denote the nodes with priority 3, square symbol to denote priority 2, and triangle symbol for priority 1. The rest are the nodes outside of the candidate set, which are assigned with priority -1. The results are presented in Figure 11. The nodes inside the candidate set, as selected by the algorithm, yielded significantly better
running a number of job replicas simultaneously [36], [22], [38], [23], [29], [34]. Usually, these efforts involve a primary execution environment and a specific class of applications. For the current checkpoint. Furthermore, LSS can be taken from Bronevetsky et al. focuses on taking checkpoints at the application level for parallel programs [17], [18], [16]. Their approach investigates the use of compiler technology to instrument codes to enable self-checkpointing and self-restarting. As we stated earlier, our work can be viewed as an optimization and adaptation of this work to a different execution environment and a specific class of applications. Furthermore, our solution is based on a middleware, as opposed to the use of a compiler.

Much work has been done on achieving fault-tolerance by running a number of job replicas simultaneously [36], [22], [38], [23], [29], [34]. Usually, these efforts involve a primary task and other backup tasks. In the case of failure on the primary task, processing continues on one of the backups. Our work achieves better resource utilization than these approaches, though it specifically targets pipelined or stream processing applications. Considering the checkpointing-based approaches, some efforts have been taken to reduce the size of checkpoints. Zheng et al. [39] have proposed an in-memory double checkpointing protocol for fault-tolerance. Without relying on any reliable storage, the checkpoint data, which is encapsulated by the programmer, is stored at two different processors. Also, the checkpoints are taken at a time when the application memory footprint is small. Another approach proposed by Marques et al. [28] dynamically partitions objects of the program into subheaps in memory. By specifying how the checkpoint mechanism treats objects in different subheaps as always save, never save and once save, they reduce the checkpoint size at runtime. Our work differs in a way that LSS is pre-defined so that there is no overhead choosing data for the current checkpoint. Furthermore, LSS can be taken with high frequency to minimize the information loss when a fault occurs, with low overhead. Also, because LSS is a logical structure, it can enable failure-recovery in heterogeneous environments.

Balazinska et al. have developed Borealis [23], [29], which is a system supporting fault-tolerance in distributed data stream processing. They proposed a Delay, Process, and Correct (DPC) approach to achieve a trade-off between system availability and consistency. DPC is also based on replication. Besides not using replication, our work is distinct in the following ways. First, we are interested in approximate processing of data streams, and thus, the results are approximate, and not completely accurate. But, there is no delay in generating the results using our approach. Second, the size of buffers can increase indefinitely in their approach, while we restricted the size of data backup buffer. Finally, Borealis reprocesses the stored tuples, while in our approach, there is no re-computation involved by using LSS.

In the area of stream processing for grids, the work that is probably the closest is the dQUOB project [32], [33]. This system enables continuous processing of SQL queries on data streams. Data stream processing has also received much attention in the data-base community as well [20]. Prominent work in this area has been done at Stanford [4], Berkeley [8], Brown and MIT [7], Wisconsin [37], among others. The focus in this community has largely been on centralized processing of a single data stream. Our focus is quite different, as we consider distributed processing of distributed data streams, and use grid resources and standards. Aurora is a framework for distributed processing of data streams, but only within a single administrative domain [12].

Earlier, it has been shown how GATES can support process migration to deal with changes in availability of CPU cycles or network bandwidth [11], but not for handling complete failure of a processing node.

VIII. Conclusion

This paper has considered the problem of supporting and efficiently implementing fault-tolerance for tightly-coupled and pipelined applications, especially streaming applications, in a grid environment. We have provided an alternative to basic checkpointing and used the notion of Light-weight Summary Structure (LSS) to enable efficient failure-recovery. Our implementation and evaluation of LSS based failure-recovery has been in the context of the GATES (Grid-based AdapTive Execution on Streams) middleware. We have shown how we perform failure-recovery and also have demonstrated how we could use additional buffers to limit data loss during failure-recovery. We have presented an efficient algorithm for allocating a new computation resource for failure-recovery at runtime.

We have extensively evaluated our implementation using three data streaming applications. The main observations from our experiments are as follows. First, the size of LSS is almost two orders of magnitude smaller than the total memory required by the application and the middleware, which points to the efficiency of our approach. Second, the overhead of storing and copying LSS in our middleware is almost negligible. The cost associated with failure-recovery is also modest. This cost increases somewhat in the version where additional buffering is done to limit data loss, or when node for failure-recovery has to be identified at runtime. Similarly, the loss of accuracy is very small, and is further reduced when data loss is limited. However, it increases if the node for failure-recovery needs to be identified at runtime. Finally, we have shown that our method for choosing a new computation resource at runtime
is effective, and failure-recovery at such nodes gives better performance than other possible options.

REFERENCES


