School of Computing Science,
University of Newcastle upon Tyne

Applying Low-Overhead Rollback-Recovery to Wide Area Distributed Query Processing

Jim Smith and Paul Watson

Technical Report Series

CS-TR-861

October 2004

Applying Low-Overhead Rollback-Recovery to Wide Area Distributed Query Processing

Jim Smith, Paul Watson

October 4, 2004

Abstract

It is argued that there is a significant class of pipelined large grain data flow computations whose wide area distribution and long running nature suggest a need for fault-tolerance, but for which existing approaches appear either costly or incomplete. This paper presents an approach which exploits limited input from the application layer to implement a low overhead recovery protocol for such data flow computations. Over a large range of possible data flow graphs, the protocol supports tolerance of a single machine failure, per execution of the computation, and in many cases a greater degree of fault-tolerance. The protocol is implemented within an emulation of a distributed query processing system. Preliminary performance measurements suggest that the overhead is indeed low.

keywords:data flow, fault-tolerance, measurement, query processing, rollback-recovery, wide area

1 Introduction

The suitability of data flow for computations which process a succession of inputs in pipeline fashion has long been appreciated [12]. Early work, e.g. [11, 23], sought to exploit very fine grain parallelism in special data flow architectures. Since this entails high bandwidth interconnect, later work aimed to increase the grain size, trading off some degree of parallelism for an easier realisation. This is manifested both in automated processing of special purpose data flow languages, e.g. [5], and in manual parallelization of essentially sequential code. While any larger grain approach is likely to suit a more loosely coupled architecture, such as networks of autonomous machines, manual approaches appear to be most used. A number of infrastructures support gluing together of pure functions [7, 4] and dynamic scheduling of the resulting *digraphs*. In applications, stateful *vertices*, supported in e.g. [22, 24, 13], can be required: to aggregate token values; or to meaningfully combine tokens from several streams.

The use of large grain data flow techniques has become established in database query processing [21]. Much work [25] has been done to support access to multiple distributed, even autonomous, databases, addressing particularly issues relating to heterogeneity, consistency, and availability. However, systems have tended to gather data to a central site for inter-site joins. As described in [32], the emergence of computational grids [18] provides much support

and motivation for the evolution of the more open query processing espoused in [6] where participants contribute not just data sources but also functionality and cycle providers. In such an environment, many widely distributed and autonomous resources may be combined into the execution of a particular query. Furthermore, it seems likely that the applications will often be demanding, so that resource failures may be not only likely but also costly. It is then preferable to tolerate the fault rather than throwing away the work done already unless the computational resources required for completion are not available.

As described in [33], other possible applications of the protocol include query processing over continuous streams [3] and publish subscribe systems [15]. The targetted applications are all pipelined dataflow and have requirements for wide area distribution and stateful *vertices*, yet also for fault-tolerance. This work shows that most existing rollback-recovery techniques are either inapplicable or likely to prove expensive when applied to such computations. The one existing proposal which seems promising is incomplete. This work presents a protocol which fills this gap and shows how it can be applied to an example computation.

In the following, section 2 describes related work, 3 a model for a distributed large grain data flow computation, 4 the rollback-recovery protocol, 5 its application in distributed query processing, and section 6 concludes.

2 Related Work

A number of systems aim to support use of idle CPU cycles [26, 1], by applications which can be expressed as a set of independent parallel tasks. Such systems may tolerate failure of a worker assigned an individual task and possibly transient failure or shutdown of the manager. However, if the processes of a parallel computation interact, care must be taken to avoid a failure potentially causing all surviving processes to roll back to their start [29].

A recent survey of protocols which support less ideally parallelizable applications [14] classifies them as being based either on coordinated checkpoint or logging. The former approach is often employed in a tightly coupled parallel machine, but achieving a coordinated checkpoint between many widely distributed processes is likely to be expensive. Log based protocols avoid the need to coordinate checkpoints of individual processes by logging indeterminate events, i.e. messages, but rely on checkpointing process state, all-be-it independently, to support pruning of the recovery logs. Such checkpoints must be made to a location where they can be accessed by whichever machine takes over the work of a failed machine. In a wide area, this location be a single machine which is far from many of the processes, or multiple separate, but local, machines. The protocol described here is similar to a log-based protocol, but exploits some input from the user level to obviate the need for checkpointing of process state.

Replication based support for fault-tolerance in software based data flow systems is described in [10, 28], but only for stateless *vertices*. It is possible to support replication of stateful *vertices*. The state machine approach [30] which sends each token to all replicas who execute in lockstep, and could be supported by [10], ensures low recovery latency but high overhead. In a coordinator-cohort

approach, a single replica responds to messages but copies its state to all others. Clearly the overhead of copying could be expensive if *vertex* state is large. A flux [31] consumer supports coordination with its producers so as to permit consistent transfer of state between machines, and thereby support dynamic load balancing. Supporting process replication for fault-tolerance is different however in requiring that the coordination of replica states be distributed between replica consumers and/or producer. By contrast, the protocol described here supports recovery for stateful *vertices*, yet avoids both repeated transmission of tokens and replication of *vertex* state in normal running in order to reduce overhead, at the cost of more expensive recovery.

In systems which assume pure functional *vertices* to permit dynamic scheduling of activations to processors, it is possible to preserve tokens used by an activation remote from the executing processor until the activation has safely written its result tokens. This allows for retry if the executing processor fails. Tokens might be stored thus in: *template memory* of a closely coupled machine [19]; shared file space in a network based system such as DAGMAN [37]; or the upstream *vertices* where they are created [27]. However, emulating stateful *vertices* in such systems is likely to be expensive. A development of this theme is to retain multiple tokens in an upstream *vertex* thereby allowing replay of arbitrary amounts of the computation. Marker tokens, e.g. *flow tuples* [9] can be inserted into the output stream to coordinate arrival of arrival of tokens downstream with their purging from logs upstream. This potentially allows restoration of *vertex* state, but such earlier work has not defined a protocol for purging logs. In the absence of such a protocol, it is always necessary for a recovering machine to replay the whole execution prior to the failure. This paper presents a log-based protocol, both in abstract and in application.

3 Computational Model

3.1 Data Flow

Figure 1 shows a partitioning of the software in a *vertex* and the mapping onto machines of *vertices* and interconnecting *edges*, which behave like FIFO queues, in an example data flow graph. A *vertex* contains some arbitrary ap-

(a) Example graph. (b) A single vertex.

Figure 1: A computational model for large grain data flow.

plication processing which can transform, generate or delete tokens, based on those it receives via certain *end points*. The application can direct result tokens to subsequent *vertices* via other *end points*. The *end points* are represented as an array of objects whose operations call on the services of an underlying infrastructure layer.

This computational model adheres to the common notion of large grain data flow in [22, 24, 13]. The application code in such a system has no access to updateable memory shared between *vertices*. However, *vertices* are statically scheduled to machines, so the application code in a *vertex* can exploit local memory, e.g. to aggregate token values. Typically for such large grain data flow, there is no set firing rule; instead a firing policy is defined implicitly, as calls on appropriate *end points*, in the application code of each *vertex*.

The structure of the application code of a *vertex* is shown in figure 2. Each

```
app() {
    do {
        // call receive() on end point(s) to get next token
        process(that−token); // do any local processing
        // output all result tokens by calling send() on end point(s)
    } while (! finished_processing());
}
```

Figure 2: Main loop in application.

loop iteration retrieves a single token, performs local processing and outputs all consequent result tokens. The choice as to which *edge* to receive a token from is defined within the application code; if required the underlying call on the Infrastructure will block. It is assumed that the application code within a *vertex* is deterministic, so that it will perform identically, independent of the order in which tokens arrive on incoming *edges*. The interface exported by an *end point* can be characterised by the following operations.

send(input Token) is called by the application code to transmit a token to the *end point* at the opposite end of the edge from this one.

receive(): returns Token is called by the application to retrieve the next available token from a particular *end point*.

handle(input Token) is called by the infrastructure on arrival of a token destined for this particular *end point*.

3.2 Fault-Tolerance

It is assumed that loss or corruption of individual messages is masked in the infrastructure service, e.g. through use of a reliable transport such as TCP [36]. Buffering of tokens in transit through restart of their destination necessitates flushing during recovery.

It is assumed that machine failures, e.g. due to power failure or reboot, are detected within the infrastructure layer. Under the most relaxed assumptions regarding the environment, as expected in a wide-area context, perfect failure detection is impossible [17]. However, it is possible to achieve consensus using unreliable failure detectors [8] and implementations of infrastructure level fault-detection services for the wide-area context have been proposed, e.g. [35].

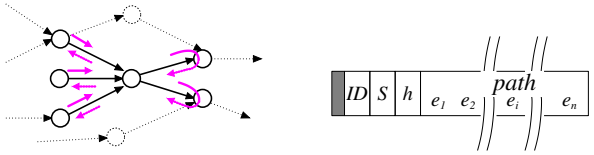
It is assumed that a replacement machine can be found, and integrated into the system by the underlying infrastructure. A number of standby machines might be included or a replacement machine found dynamically when needed. The choice between such policies is likely to be influenced by cost and scale of distribution, but is not of concern here. Integration of a standby to replace a failed machine can be seen as ensuring that for each surviving machine, the mapping between logical participant and physical machine identification is updated. Consistent detection and integration can be achieved, for instance, in the well known message passing infrastructure MPI [34], e.g. [16].

The focus of the work described here is to ensure correct behaviour of the application given appropriate prompts by the system level support. In particular, having (re)integrated with a computation, the infrastructure initiates the local application code. The restart part of the rollback-recovery protocol is initiated automatically at this time.

Machines are not required to have stable storage; buffering employed by the rollback-recovery protocol can take place in volatile memory which is initialised at (re)start. Where space restrictions necessitate spooling recovery log data onto local disk, such spooled data can similarly be discarded in restart.

4 The Rollback-Recovery Protocol

A fuller description of the rollback-recovery protocol appears in [33], but the main results are summarised here. The protocol relies on the return of acknowledgements from downstream *vertices* to support the truncation of a recovery log which contains tokens sent out along a particular *edge*. Acknowledgement of a *block* of data tokens entails return of a checkpoint marker inserted into the data flow after those data tokens by an upstream *end point* which sent them, as shown in figure 3(a). A checkpoint marker, see figure 3(b), carries a collection



(a) A *segment* of a *slice*. (b) Checkpoint marker.

Figure 3: Elements of the rollback-recovery protocol..

of values, *path*. This collection is actually a stack into which each incoming *end point* pushes its *ID* as the checkpoint marker passes through on its forward journey. This *path* supports relay of an acknowledgement to the creator of its corresponding checkpoint marker. A checkpoint marker is returned thus when it has travelled over a given number of *edges*, e.g. 2 in figure 3(a). This technique is suited for *uniform digraphs* which satisfy the property that there are no two vertices connected by two *paths* of different length [33].

Each checkpoint marker carries a sequence number S which is unique for its creating *end point*. An *outgoing end point* increments its sequence number

with each checkpoint marker creation. When an acknowledgement arrives at the source of its corresponding checkpoint marker all tokens preceding that checkpoint marker are discarded from the recovery log there. An *incoming end point* buffers received tokens, only releasing them for access by the application code via *receive()* when a checkpoint marker arrives, whose sequence number S is more recent than the latest received from the same *ID*. If the central *vertex* fails and is replaced, a *restart* request is sent to each upstream adjacent *vertex*, i.e. *source* in the *segment*, at which that *vertex* first flushes the connecting edge and then replays the contents of its recovery log, including checkpoint markers. The application code in the *central vertex*, being deterministic, creates exactly the same stream of output tokens as it did first time. *Incoming end points* in the sinks of the *segment* purge any partial block of tokens as recovery starts, and discard duplicate blocks received during recovery.

As described in [33], it is possible to make an arbitrary *uniform digraph* fault-tolerant by overlapping multiple *slices*, but not 1-fault-tolerant, because in general the earliest tokens which a failing vertex had been backing up for a next but one downstream vertex are not re-created in the recovery process, having been acknowledged by the downstream adjacent *vertices*. To achieve 1-fault-tolerance requires acknowledgements to be sent from the *sinks* of the *digraph*. Constraining the protocol within a *slice* trades off a measure of fault-tolerance to limit recovery log size and acknowledgement distance.

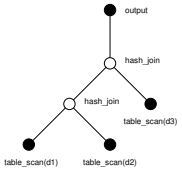
An *incoming* end point requires to buffer up to one block of tokens and remember the latest checkpoint marker from each upstream *end point* whose checkpoints it handles. The main buffering requirement lies in an *outgoing end point* which needs to buffer as many tokens as will be unacknowledged at any given time. This quantity is determined partly by the selected block size and partly by application characteristics.

As described in [33], most processing related to the rollback-recovery protocol can be encapsulated in the *end point* operations *send()* and *handle()*. However, the application code in a *vertex* is responsible for: forwarding checkpoint marker tokens; and relaying acknowledgements. In the former case, the application code forwards a checkpoint marker when it knows the corresponding tokens are not required any more. In the latter case, the application code simply ensures that an acknowledgement is only sent upstream when it has been received from each downstream adjacent *vertex*. While the former requirement is quite application specific, the latter is actually quite generic.

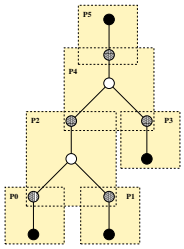
5 Application to Distributed Query Processing

A common, and potentially demanding, requirement in distributed query processing is to join multiple data sources using one or more attributes common to each pair of sources. Often, further processing is applied to other attributes, but here such further processing is ignored. Typically in a query processing system, an optimiser employs data related statistics to select a physical query execution plan, a directed graph whose vertices are operators of the physical algebra, e.g. hash join etc, and edges are data paths between operators. A distributed

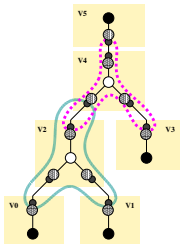
query processing system can enhance its algebra through the addition of an extra operator, often termed exchange, which encapsulates communications and execution thread related concerns, and add parallelization and scheduling steps to the query optimization process to select a parallel query execution plan [32]. A pair of exchange operators placed in separate partitions appear just like any other operator as far as the rest of the query plan is concerned yet manage internally the efficient transfer of tuples between distinct machines hosting the partitions. Figure 4(a) shows one possible plan, a *left-deep-tree*, which might be selected as the physical plan for a three-way natural join. Figure 4(b) shows a



(a) Physical plan.



(b) Parallel plan, including exhanges.



(c) Recovery protocol slices.

Figure 4: Mapping an example query onto computational resources.

possible parallel query plan where, following [32], a pair of exchange operators marking the border between partitions is represented by a single instance with the corresponding partitions overlapping. The decision to locate each join on a separate machine from either of those hosting one of its inputs might be made either: due to administrative restrictions; or because the remote machine, unlike either host, has sufficient main memory to permit use of a single pass join. Figure 4(c) equivalently shows the individual exchange operators within separate partitions, in order also to show the *end points*, and how the rollback-recovery protocol can be employed, in two overlapping *slices*.

Figure 5, shows how the hash join operator employed in this query plan can be modified to forward checkpoint markers. The left input is retained entire in a hash table while the right input is read, so while checkpoint markers are forwarded directly from the right input, only the last read is forwarded from the left input, at the end. After a failure, the whole of the left input is replayed, but this is typically much smaller than the right input or such an operator would not have been chosen. Only that part of the right input which has not been acknowledged at the time of the failure is replayed.

The example doesn't show a redistribution of tuples from one machine to many, but such a redistribution would be encapsulated in an exchange operator in the producer vertex. This encapsulation offers an ideal spot to locate the coordination of acknowledgements. In the example shown here, an *outgoing end point* can direct each acknowledgement it receives directly to the *incoming end point* indicated in the *path* in the acknowledgement.

For initial evaluation, a distributed query processing system has been emulated in a uni-process, but multithreaded Java applet. A set of physical opera-

```
public void open() {
    left.open(); t = left.next(); while (! t.iseof()) {
        if (t.ischeckpoint()) llatest = (CheckpointMarker) t; else
            h.insert((DataTuple)t); t = left.next();
    } left.close(); right.open(); }                                    5
public Tuple next() {
    if (!results.empty()) return (Tuple) results.pop();
    if (eof) return new EOFTuple(); else t = right.next();
    while (true) {
        if (t.iseof()) {eof = true; return llatest;}                   10
        if (t.ischeckpoint()) return t;
        results = h.probe((DataTuple)t);
        if (results.empty()) t = right.next();
        else return (Tuple) results.pop(); }}
```

Figure 5: Changing a single phase hash join operator (inserted lines 3,10,11) to support checkpoint marker processing.

tors is implemented, similar to those of [2]. The operators here are simplified versions, but much of the code implementing the recovery protocol has ported directly into the full system in the ongoing implementation there.

In the evaluation system the exchange is multi-threaded as in [2], but the inter-service call is replaced by a simple object method call as the whole evaluation runs in a single process. While this simplification avoids the complexity of true distribution, it tends to emphasize the overhead cost of the recovery protocol, since the cost of converting buffers of tuples to and from SOAP messages and transmitting these messages across the wire in the full system affects both data and protocol messages alike, and there are typically more of the former. The 'table scan operator employed here is also a simplification of that in the full system. Rather than accessing data from an external database, such as a MySQL server, the table scan employed here is parameterized, via the query plan input, to generate tuples. Generated tuple attributes can be defined explicitly, as in a list of colours, or implicitly, as in a sequence of stringified numbers having minimum and maximum values and increment. Again, the greatly reduced cost of generating tuples can only be expected to emphasize the overhead cost of the recovery protocol. Rather than employing the optimization system of [2] to generate a query plan from a query language such as OQL, the evaluation system takes as input a hand coded plan divided into parts which are to be scheduled to separate *vertices*, i.e. machines in a distributed setting. The creation of *vertices* and installation of query plan partitions is similar to that performed in the full system, but simplified by the *vertices* here being pure Java objects rather than Web Services.

In the evaluation system, low level failure management only requires destroying (resetting pointers to) an instance of the Java object *Vertex*, and subsequently installing a new instance, e.g. in response to mouse-click. To facilitate repeatable experiments, query execution can optionally be slowed by inserting a short pause in communication of a tuple between vertices.

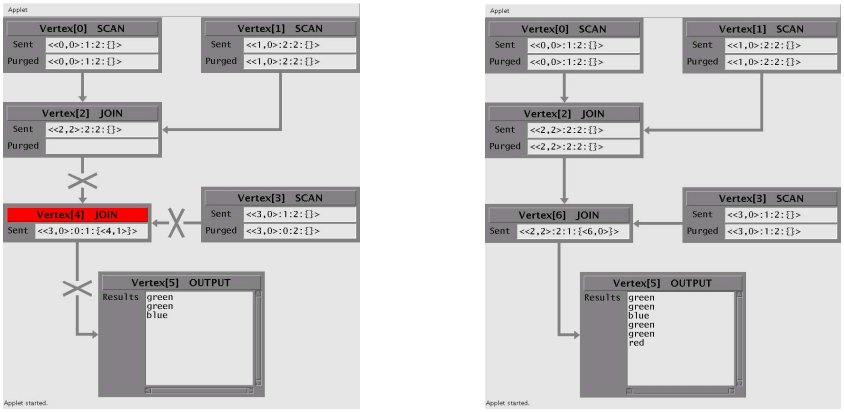
As a concrete example, the left deep tree plan described earlier is presented,

combining three data sets which have a single attribute each, "colour"; effectively three lists of colours are joined. The actual data sets, shown in figure 6, are tiny but still permit demonstration of the rollback-recovery protocol, through the slowed execution and reduced protocol block size of 4.

data set	values
d1	green, red, yellow, green, blue, black
d2	red, purple, yellow, black, green, purple, white, yellow, blue
d3	white, green, blue, purple, green, orange, red

Figure 6: Three tiny data sets.

Figure 7 shows two screen dumps of the demonstration applet during an example run. In figure 7(a), *vertex* 4, containing the second join has been failed. *Vertex* 2, the left input to the failed join, had completed its data generation; the recovery log in its output contains all tuples it had generated, unacknowledged. At the time of failure, *vertex* 4 was processing the last block of tuples generated by *vertex* 3; the last tuple sent by *vertex* 3 is a checkpoint marker with sequence 1 and hop count 2. When *vertex* 4 is replaced by the new *vertex* 6, figure 7(b), the recovery logs in both *vertices* 2 and 3 are replayed, allowing the join in *vertex* 6 to complete the missing output, whereupon acknowledgements are sent to *vertices* 2 and 3. Recovery is a benefit here since there was no need to replay the early part of the query, involving *vertices* 0 and 1, nor the early outputs of *vertex* 3 which had been acknowledged by the time of the failure. In a realistic example, the data sets output by *vertices* 1 and 3 could be large. Extensive testing with the applet shows the protocol can support failure and



(a)A while after vertex 4, containing the second join, has failed.

(b)After vertex 4 has been replaced, by vertex 6, and the run completed.

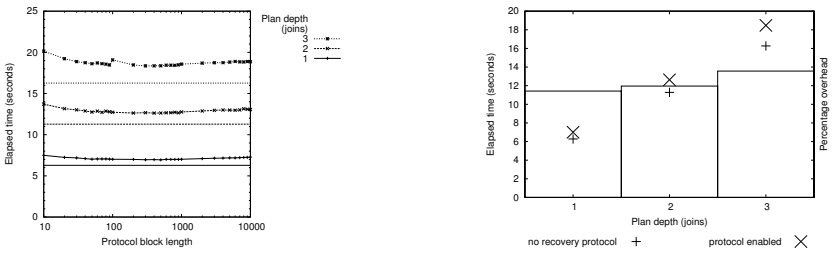
Figure 7: Protocol demonstration.

replacement of *vertex* 2 or 4 at any point. In the case of query processing over persistent data, a slight enhancement supports failure and replacement of one

of *vertices* 0,1,3. A replacement for such a leaf *vertex* reuses the same *ID*, so that when it repeats the scan, it re-creates identical checkpoint markers, so a downstream adjacent *vertex* can recognise duplicates.

The same evaluation system can be used outside of the applet framework to conduct performance measurements. In the following experiments left-deep-trees of one and three joins, combining respectively two and four datasets are used in addition to the earlier example. For ease of analysis, each dataset is identical, comprising 500000 10 byte tuples, each having as single attribute a unique decimal value with leading spaces. Each hash join combines a pair of such datasets to yield an identical dataset as result. In this way, the cost of processing the query should scale linearly with the number of joins.

The following measurements were made on an IBM thinkpad model T40 with 775 MB memory and 600 MHz Intel processor. Figure 8(a) that over the



(a)Overhead vs block size. (b)Minimum cost per plan.

Figure 8: Protocol overhead.

range of queries, the overhead cost of the recovery protocol falls to a minimum at a block size of 200. The end point implementation uses local buffers to increase the granularity of access to buffers which are shared between separate threads, and thus incur synchronization overhead. In these experiments the size of these internal buffers was set uniformly to 1000. While the overhead cost is clearly expected to fall as the block size is increased, it is the fixed size of these internal buffers that limits that fall. Figure 8(b) shows the protocol implementation scales well, being effectively a fixed percentage of the overall query cost over the range of query plans. Since each additional join brings an identical extra dataset, this is to be expected. As described earlier, by comparison with a true distributed query processing system, the evaluation system has greatly reduced communication and data access costs, which tend to emphasize the protocol cost. In this context, the absolute value of the protocol overhead, at about 12% must be seen as small.

6 Conclusions

It has been argued that there is a class of applications which naturally suit a pipelined large grain data flow expression, but which through being long running and distributed over autonomous resources in a wide area, require provision

for fault-tolerance. General purpose approaches to fault-tolerance seem likely to incur a high cost, particularly in a wide area context, e.g. through checkpointing potentially large state to remote sites. Data flow specific approaches include replication and log based techniques. While the former can support fast recovery from multiple failures, the cost in terms of extra communication and or processing seems likely to be high, again particularly in a wide area context. The latter seems intuitively to incur low overhead, but existing proposals are incomplete, particularly in their provision for pruning the recovery logs. With no checkpointing of state, it is essential to provide for such log pruning to avoid indefinite roll-back in the event of a process failure.

A protocol has been presented which fills this gap for *uniform digraphs*, which have no alternate *walks* of differing length. Tokens are acknowledged a fixed distance from the *vertex* where they are generated. The protocol is then bounded within a *slice* of the graph, such that tokens are generated in its *sources* and acknowledged in its *sinks*, the maximum distance between a *source* and *sink* of a *slice* being its *thickness*. The correspondence between recovery log position and acknowledgement is established by the insertion of *checkpoint markers* into the data stream; these are returned as acknowledgements. The application of this protocol to an example distributed query is described and promising measurements of protocol overhead given.

Ongoing work is investigating: related protocols which will support a *thicker slice*; cost models to support investigation into alternative strategies for partitioning a data flow graph and a quantitative comparison with alternative fault-tolerance strategies; and issues related to exploiting a protocol such as that presented here in the context of adaptation, e.g. [20]. An implementation of the protocol within a prototype service based distributed query processing system [2] is nearing completion.

References

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A tool for performing parameterised simulations using distributed workstations. In *High Performance Distributed Computing*, pages 112–121, Washington, DC, USA, August 1995.

[2] M. N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A.A. Fernandes, A. Gounaris, and J. Smith. Service-based distributed query processing on the grid. In *International Conference on Service Oriented Computing*, Trento, Italy, December 2003. Springer Verlag.

[3] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, September 2001.

[4] A. Beguelin and J. J. Dongarra. Graphical development tools for network-based concurrent supercomputing. In *Supercomputing*, pages 435–444, Albuquerque, New Mexico, USA, November 1991. ACM Press.

[5] W. Böhm, W. Najjar, B. Shankar, and Lucas Roh. An evaluation of coarse grain dataflow code generation strategies. In *Working Conference on Massively Parallel Programming Models*, Berlin, Germany, September 1993.

[6] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Pröls, S. Seltzsam, and K. Stocker. Objectglobe: Ubiquitous query processing. *The VLDB Journal*, 10(1):48–71, August 2001.

[7] J. C. Browne, J. Werth, and T. Lee. Experimental evaluation of a reusability-oriented parallel programming environment. *IEEE Transactions on Software Engineering*, 15(2):111–120, February 1990.

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[9] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distribute stream processing. In *Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[10] R. Davoli, L-A Gianchini, Ö Babaoglu, A. Amoroso, and L. Alvisi. Parallel computing in networks of workstations with paralex. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):371–387, April 1996.

[11] J. B. Dennis. First version of a data flow language. MAC Technical Memorandum 61, Cambridge Massachusetts, May 1975.

[12] J. B. Dennis and kung Song Weng. An abstract implementation for concurrent computation with streams. In *International Conference on Parallel Processing*, pages 35–45. IEEE Computer Society, August 1979.

[13] R. E. Eggen and J. R. Metzer. An inherently parallel large grained data flow environment. In *Annual Conference on Computer Science*, pages 551–557, Atlanta, Georgia, USA, February 1988. ACM Press.

[14] E. N. (Mootaz) Elnozahy, L. Alvisi, Y-M Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[15] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A-M Kermarrec. The many faces of pulish subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[16] G. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting*, pages 346–353, Balatonfüred, Hungary, 2000. Springer.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one failure. *Journal of the ACM*, 32(2):374–382, April 1985.

[18] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Fransisco, 1999.

[19] J-L Gaudiot and C. S. Raghavendra. Fault-tolerance and data-flow systems. In *International Conference on Distributed Computing Systems*, pages 16–23, Denver, CO, USA, 1985. IEEE Computer Society.

[20] A. Gounaris, , N. W. Paton, A. A.A. Fernandes, and R. Sakalleriou. Adaptive query processing: A survey. In *British National Conference on Databases*, pages 11–25. Springer Verlag, August 2002.

[21] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[22] R. Babb II. parallel processing with large grain data flow techniques. *Computer*, 17(7):55–61, July 1984.

[23] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland publishing company, Amsterdam.

[24] I. Kaplan. The LDF 100: A large grain dataflow parallel processor. *SIGARCH Computer Architecture News*, 15(3):5–12, June 1987.

[25] D. Kossman. The state of the art in distributed query processing. *Computing Surveys*, 32(4):422–469, December 2000.

[26] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *International Conference on Distributed Computing Systems*, pages 104–111, San Jose, CA, USA, June 1998. IEEE Computer Society.

[27] W. Najjar and J-L Gaudiot. A data-driven execution paradigm for distributed fault-tolerance. In *SIGOPS European Workshop*, pages 1–5, Bologna, Italy, 1990. ACM.

[28] A. Nguyen-Tuong, A. S. Grimshaw, and M. Hyett. Exploiting data-flow for fault-tolerance in a wide-area parallel system. In *Symposium on Reliable Distributed Systems*, pages 2–11, Ontario, Canada, October 1996. IEEE Computer Society.

[29] B. Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, June 1975.

[30] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[31] M. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *International Conference on Data Engineering*, Bangalore, India, March 2003. IEEE Computer Society.

[32] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A.A. Fernandes, and R. Sakalleriou. Distributed query processing on the grid. In *International Workshop on Grid Computing*, pages 279–290, Baltimore, MD, USA, November 2002.

[33] J. Smith and P. Watson. A rollback-recovery protocol for wide area pipelined data flow computations. Technical Report CS-TR-836, University of Newcastle upon Tyne, May 2004.

[34] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference*. The MIT Press, Cambrideg, MA, 1998.

[35] P. Stelling, C. DeMatteis, I. T. Foster, C. Kesselman, C. A. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.

[36] A. S. Tannenbaum, editor. *Computer Networks*. Prentice-Hall, 1991.

[37] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, A. Hey, and G. Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 299–335. John Wiley and sons Ltd., 2003.