School of Computing Science,
University of Newcastle upon Tyne

A Rollback-Recovery Protocol for Wide Area Pipelined Data Flow Computations

Jim Smith and Paul Watson

Technical Report Series

CS-TR-836

April 2004

Copyright©2004 University of Newcastle upon Tyne
Published by the University of Newcastle upon Tyne,
School of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK.

A Rollback-Recovery Protocol for Wide Area Pipelined Data Flow Computations

Jim Smith and Paul Watson

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. An example, which motivated this paper, is the execution of queries over distributed databases. This paper presents an approach which exploits some limited input from the application layer in order to implement a low overhead recovery protocol for such data flow computations. Over a large range of possible data flow graphs, the protocol is shown to support tolerance of a single machine failure, per execution of the data flow computation, and in many cases to provide a greater degree of fault-tolerance.

Index Terms

data flow, distributed system, fault-tolerance, parallel system, rollback-recovery, wide area

I. Introduction

The suitability of data flow for computations which process a succession of inputs in pipeline fashion has long been appreciated [16]. Early work, e.g. [15], [17], [28], sought to exploit very fine grain parallelism in special data flow architectures. However, this entails high bandwidth interconnect, so later work aimed to increase the grain size, trading off some degree of parallelism for an easier realisation. This trend manifested itself both in automated processing of special purpose data flow languages, e.g. [8], 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. There are a number of infrastructures which support gluing together of pure functions [10], [5] and dynamic scheduling of the resulting *digraphs*. Within applications however, the support for stateful *vertices* as offered in systems such as [27], [30], [18] is often assumed, for instance to aggregate several 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 [26]. Much work [31] 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. By contrast [9] makes a case for a more open form of distributed query processing where participants contribute not just data sources but also functionality and cycle providers. As described in [38], the emergence of computational grids [23] provides much support and motivation for the evolution of the sort of more open query processing espoused in [9]. In this open environment, many widely distributed and autonomous resources may be combined into the execution of any 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. In such situations, it would be preferable to tolerate the fault rather than

Jim Smith is with the School of Computing Science, Newcastle University, UK
Paul Watson is with the School of Computing Science, Newcastle University, UK

throwing away the work done already unless the computational resources required for completion were not available.

Much recent work in the area of data management has been focused on query processing over continuous streams [2]. While there is obviously some overlap with data set processing, stream query processing enables new types of user level queries and requires new approaches to query execution. Obvious queries include the running average of a selection of items, the identification of the highest values over a period of time, and the identification of all changes greater than some threshold of certain items. Emerging operators promise support for queries which are more complex and which combine multiple streams, e.g [29], [13], [44]. The volume of real-time data available is likely to increase greatly and users are likely to seek to enter progressively more complex queries over it. Typical stream processing systems may access widely distributed data, and may employ parallelism [37], but are essentially centralized. By contrast [43] anticipates query plans being distributed over multiple sites, perhaps linking together stream resources made available by separate organisations. In such a context the requirement for fault-tolerance obviously arises again, perhaps with a greater urgency.

Publish subscribe systems [20] are long running and manipulate streams of events, distributing them over a wide area to a potentially very large number of subscribers. Both the dissemination and the filtering may be distributed over a number of sites [4] and there is suggestion that testing for the correlation of multiple events may be desirable [3]. One strategy to meet the requirements for supporting asynchrony would be to maintain the necessary storage of events at the leaves of the tree, which would be geographically closer to the subscribers. The non-leaf nodes would then form a simple wide area data flow graph.

Such pipelined dataflow applications have requirements for wide area distribution and stateful *vertices*, yet also have requirements 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. To address this gap, a large class of *digraphs*, which seem likely to be the most commonly used, is identified. These *digraphs* have properties that inspire a family of protocols which can exploit limited input from the application level to provide low overhead fault-tolerance support. The work goes on to define and verify a detailed proposal for the simplest of these protocols.

The rest of this paper is structured as follows. Section II discusses related work. Section III defines a model for a distributed large grain data flow computation. Section IV presents the rollback-recovery protocol in the context of this model and establishes the correctness of its behaviour and section V concludes.

II. Related Work

A number of systems aim to support use of idle CPU cycles [32], [1], by implementing management for applications which can be expressed as a set of independent tasks. Such systems typically support tolerance of 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 need to interact, then care must be taken in fault-tolerance support to avoid a failure potentially causing all surviving processes to roll back to their start [35].

A recent survey of protocols which support less ideally parallelizable applications [19] classifies them as being based either on coordinated checkpoint or logging. Coordinated checkpointing 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. However, they all rely on checkpointing process state, all-be-it independently, in order 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 context, checkpoints could be copied to a single machine which is located far from many of the processes, or to multiple separate, but local, machines. The protocol described here is similar to a log-based protocol, but exploits some additional 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 [14], [34], but only for stateless *vertices*. While [14] builds on the group messaging services of ISIS [6], [7], [34] uses a less reliable but cheaper message infrastructure. It is possible to support replication of stateful *vertices*. The state machine approach [36] which sends each token to all replicas who execute in lockstep, and could be supported by [14], 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 [37] 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.

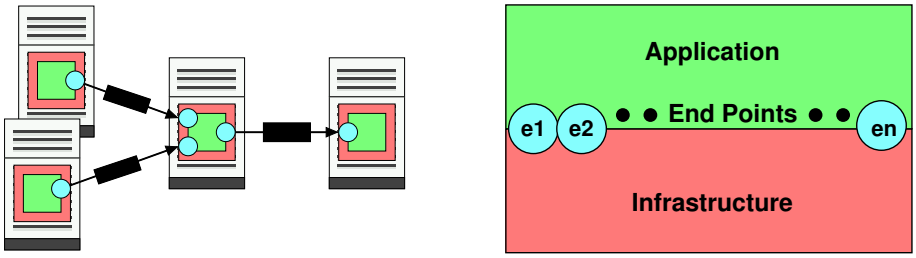
In systems which assume pure functional *vertices* to permit dynamic scheduling of activations to processors, it is possible to preserve tokens used by an activation remote from the executing processor until the activation has safely written its result tokens. This allows for retry if the executing processor fails. Tokens might be stored thus in: *template memory* of a closely coupled machine [24]; shared file space in a network based system such as DAGMAN [42]; or the upstream *vertices* where they are created [33]. 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* [12] 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 complete log-based protocol and establishes its correctness properties.

III. Computational Model

A. Data Flow

This section defines a framework which supports statically scheduled large grain dataflow computations. The rollback-recovery protocol will be presented in the context of this framework.

Figure 1 shows a partitioning of the software in a *vertex* and the mapping of *vertices* and interconnecting *edges* in an example data flow graph onto machines. A *vertex* contains some arbitrary application 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.



(a) An example graph. (b) A single vertex.

Fig. 1. Architectural layers and mapping in a computational model for large grain data flow.

The computational model defined above adheres to the common notion of large grain data flow in [27], [30], [18]. In common with the framework proposed here, the user of one of these systems has no access to updateable memory shared between *vertices*. However, *vertices* are statically scheduled to machines, thereby permitting the application code in a *vertex* to 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 application code of a *vertex* is represented as shown in figure 2, where each loop iteration

```
app() {
    do {
        Token t = get_next(); // i.e. call receive() on
                              // whichever end point(s)
                              // to get next token
        process(t);           // do any local processing
        put_any();            // output all result
                              // tokens by calling send()
                              // on some end point(s)
    } while (! finished_processing());
}
```

Fig. 2. Main loop in application.

retrieves a single token, performs local processing and outputs all consequent result tokens. As indicated earlier, the application code in a *vertex* is not triggered by a globally defined firing rule. The choice as to which *edge* to receive a token from is assumed to be defined within the application specific operation *get_next()*; if required the underlying call on the Infrastructure will block. It is then assumed that the order in which the application code within a *vertex* processes is deterministic, so that the application code in a *vertex* will perform consistently, 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 which is destined for the *end point* at the opposite end of the edge terminating locally in the corresponding *end point*.

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

```
send(Token t) {
    dosend(t); // ... to infrastructure layer
}
```

Fig. 3. Sending a token to the destination end point.

```
Token receive() {
    // wait if inputq empty
    return inputq.dequeue();
}
```

Fig. 4. Getting the next available token.

particular *end point*. It is convenient in this presentation to assume the availability of an *up-call* from the infrastructure level as indicated here, and such an up-call is likely to minimize delay in response to incoming data. In the absence of such an up-call however, the processing performed by *handle()* could be partitioned between the *send()* and *receive()* operations, using whatever operations are available in the infrastructure interface to access tokens buffered there.

```
handle(Token t) {
    inputq.enqueue(t);
}
```

Fig. 5. Processing a token delivered by the infrastructure layer.

dosend(input Token) is a helper function which encapsulates a call on the infrastructure layer to transfer the token passed as a parameter to the other end of the *edge* associated with this *end point*.

B. 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 [41]. Further, it is assumed that *dosend()* passes its parameter to the infrastructure through an asynchronous call and that the token may be buffered through restart of the destination.

It is also 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 known to be impossible [22]. However, it is possible to achieve consensus using unreliable failure detectors [11] and implementations of infrastructure level fault-detection services for the wide-area context have been proposed, e.g. [40].

It is further assumed that a replacement machine can be found, and integrated into the system by the underlying infrastructure. One policy is to include provision of some number of standby machines. Another is to rely on dynamic acquisition of a machine 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 (at least those that will need to communicate with the new one), 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 [39], e.g. [21].

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 computa-

tion, 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.

IV. The Rollback-Recovery Protocol

A. Preliminary

The protocol described in this document relies on the return of acknowledgements from downstream *vertices* in order to support the truncation of a recovery log which contains tokens sent out along a particular *edge*. Let D be a *digraph* with *vertices* $vertex(D)$ and *edges* $edge(D)$. Figure 6 shows the propagation of tokens via a single *edge* from a particular *vertex* v_0 within D, and the return of acknowledgements from downstream *vertices* to v_0. v_0 maintains copies of these tokens in

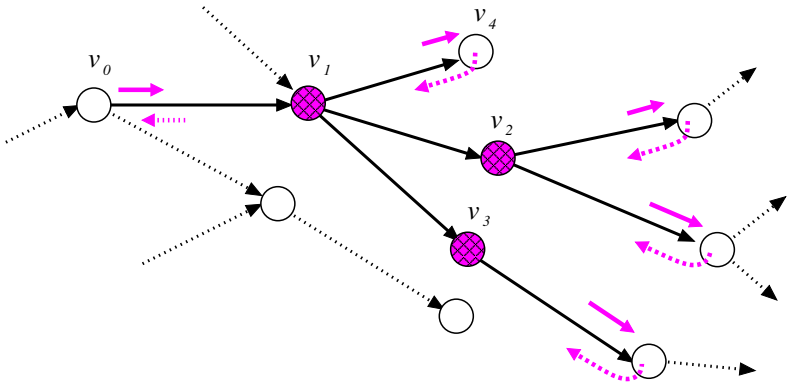


Fig. 6. Propagation of tokens and acknowledgements in an example graph.

a recovery log until they are acknowledged. The aim of the rollback recovery protocol is to support tolerance to failure of *vertices* lying between v_0 and those *vertices* downstream which return the acknowledgements, e.g. $\{v_1, v_2, v_3\}$. In general the subgraph traced out in this way may include one or more *sink vertices* of D, such as v_4.

Let d be a *digraph* which has $source(d) = \{v_0\}$ and $\forall v \in vertex(d)(distance(v_0, v) \leq h)$, where h is a positive integer. Let $walk(u, v)$ be the possibly empty set of *walks* between u and v where $u, v \in vertex(d)$.

- If d is cyclic, it is possible to walk around a cycle. Therefore $cyclic(d) \rightarrow \exists w \in walk(v_0, v)(|w| > h)$.
- Assume that d is *acyclic*, but there is a walk longer than h. It follows that two *vertices* are connected via unequal length walks, $\neg cyclic(d) \wedge (\exists u, v \in vertex(d)(\exists w \in walk(u, v)(|w| > h))) \rightarrow \exists u, v \in w(\exists w_1, w_2 \in walk(u, v)(|w_1| \neq |w_2|))$.

Let a *digraph* D be *uniform* iff

$$\forall u, v \in vertex(D) \forall w_1, w_2 \in walk(u, v)(|w_1| = |w_2| = distance(u, v)).$$

Intuitively, non-*uniform* data flow graphs, having loops and/or asymmetries, are likely to be hard to manage in execution. Accordingly, a planning agent, such as a query optimizer, would try to avoid them. In the first instance then, attention is restricted to *uniform* graphs.

Let D be a *uniform digraph* and let $d = subg(D, u, h)$ be a subgraph of D having $source(d) = \{u\}$, $\forall v \in vertex(D)(distance(u, v) \leq h \rightarrow v \in vertex(d))$ and $\forall e \in edge(D)(head(e) \in vertex(d) \wedge tail(e) \in vertex(d) \rightarrow e \in edge(d))$. Clearly, d is *uniform*, so the operation *subg* returns a *uniform digraph*. In such a subgraph d, if the protocol ensures that acknowledgements are returned when tokens output by the *source* v_0 have propagated a distance h or reached a *sink* (of D), then the *vertices* where acknowledgements are generated are only those in *sink(d)*. It follows that the protocol can ensure some measure of fault-tolerance, to be defined later, for all *vertices* $v \in inner(d) = vertex(d) \setminus \{v_0\} \setminus sink(D)$.

A token can only be purged from the recovery log in v_0 when it has been acknowledged by all *vertices* at distance h from v_0 which should receive that token. While it is permitted for a *vertex* to copy a token to an arbitrary number of output *edges*, it is convenient to place a tighter restriction on the handling of acknowledgements. Specifically, a token is only purged from the recovery log when it has been acknowledged by all *sinks* of d. This condition can be satisfied either by having all such *vertices* send acknowledgements directly to v_0, or by having acknowledgements relayed via the intervening *vertices*, and combined there. In the former case, each *vertex* has to "know" how many acknowledgements to expect. In the latter case, each *vertex* only has to know how many out-going *edges* it has. For this work, the latter option is assumed.

Let P be the set of all subgraphs $p = subg(D, u, h_0)$ where h_0 is a constant. Let $root(p)$ be a function that returns the single *source* of p. Let Q be a set of such subgraphs which are all connected via their *sinks*, $\forall p \in P \left(sink(p) \cap \left(\bigcup_{q \in Q} sink(q)\right) \neq \Phi \leftrightarrow p \in Q\right)$. The union of all subgraphs in Q can be thought of as a *slice* of D. In a *slice*, each *vertex* is a constant distance from all *sinks* it is reachable from. Specifically, the length of each *walk* between a *source* and a *sink* within a *slice* is equal to h_0. Consequently, this length can be thought of as the *thickness* of the *slice*.

As elsewhere [19], support for tolerance to failure of either *source* or *sink vertices* of D which entails coordination with the real world requires some further mechanism. One possibility is for a *source* to log all input received onto stable storage before processing it and for a *sink* to save each result onto stable storage before outputting it.

B. Checkpoint Marker Tokens

Acknowledgement of data tokens is accomplished by *end points* inserting checkpoint marker tokens into their output streams. Each checkpoint marker carries a sequence number which is unique for its creator. An *end point* increments its sequence number with each checkpoint marker creation. Tokens lying between a pair of checkpoint markers can be thought of as a *block* marked by the checkpoint marker immediately following. To acknowledge receipt of all tokens up to a given checkpoint marker, it is only necessary to return the checkpoint marker itself.

Figure 7 shows a possible structure for such a checkpoint marker. The shaded box represents a

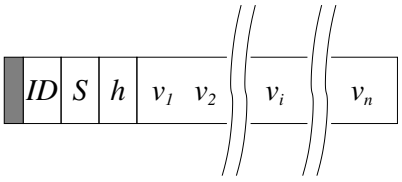


Fig. 7. A checkpoint marker token.

field which distinguishes this token as a checkpoint marker. *ID* is a value which may be used to

distinguish the creator, i.e. *end point* of this checkpoint marker. This value is constant during the participation by the corresponding *vertex* within a data flow computation. It is employed within *end points* of downstream vertices to identify which saved checkpoint marker a newly received one should be compared with; an *end point* need only keep the latest checkpoint marker it receives from any upstream *end point*. An integer index could be assigned centrally to a *vertex* when it joins a computation, and *end points* numbered sequentially within a *vertex* to yield a suitable *ID*. *S* is the sequence number value. *h* indicates the number of hops (i.e. *vertices*), the checkpoint marker should travel downstream before it is acknowledged. The entries *v1*... *vn* represent a list of *vertices* defining the route taken by the checkpoint marker in its forward path. As described earlier, the list is traversed in the opposite direction as the checkpoint marker retraces that path to its originator as an acknowledgement. Within a *slice*, checkpoint markers are generated and acknowledged respectively in *sources* and *sinks* of that *slice*.

C. Rollback-Recovery in a Slice *of* thickness *2*

If the *slice thickness* is 2, then a pair of subgraphs within a slice can only share a *source* and or a *sink*. Consequently, the protocol is bounded within a simpler graph, comprising a single *central vertex* and a number of *sources* and *sinks*. An example of such a *segment* of a *slice* of *thickness* 2 is shown in figure 8.

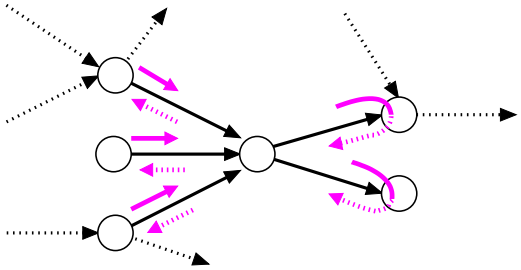


Fig. 8. Scope of the rollback-recovery protocol in a *slice* of *thickness* 2.

C.1 Protocol Operation in a Single Slice

Figure 9 shows the generation of checkpoint markers encapsulated in the *end point send()* operation. When the local counter *tcount* reaches a set value, a new checkpoint marker is generated

```
send(Token t) {
    if (++tcount > BLOCKSIZE()) {
        tcount = 1;
        // set hop count to 2
        cm = new checkpoint_marker(sequence++,2);
        recoverylog.enqueue(cm);
        dosend(cm); // ... to infrastructure layer
    }
    recoverylog.enqueue(t);
    dosend(t); // ... to infrastructure layer
}
```

Fig. 9. Incorporating checkpoint marker generation into the *send()* operation.

using the instance variable *sequence*. All tokens and checkpoint markers are copied to the recovery log.

Figure 10 shows how checkpoint and acknowledgement tokens can be processed on arrival at an *end point*. If this *end point* is the tail of an out-going *edge*, the token received may be of two types.

```
handle(Token t) {
    if (t.kind == acknowledgement) {
        if (--t.hopcount < 0) {
            // remove tokens and corresponding checkpoint markers cm
            // from restart queue, where t.sequence >= cm.sequence
        } else {
            // app code collates acks from multiple downstream end points
            newest_ack[t.sender] = t;
        }
    } else if (t.kind == restart) {
        t.kind = flushed;
        dosend(t); // tokens in transit are flushed
        // send all tokens, including checkpoint markers, from
        // the recovery log, in log order
    } else if (t.kind == flushed) {
        flushing = false;
        // discard any entries in bufferq
    } else if (!flushing) { // flag is true at startup
        if (t.kind == checkpoint) {
            if (--t.hopcount <= 0) {
                t.hopcount = t.path.entries(); // i.e. distance travelled
                t.kind = acknowledgement;
                dosend(t);
                if (t is more recent than remembered value for this source) {
                    // copy entries in bufferq to inputq
                } else {
                    // discard corresponding entries in bufferq
                }
            } else {
                t.path.push(myid); // add me into path so ack can retrace
                bufferq.enqueue(t);
            }
        } else
            bufferq.enqueue(t);
    } else {
        // flushing, so discard tuple
    }
}
```

Fig. 10. Incorporating checkpoint marker and acknowledgement processing into the *handle()* operation.

acknowledgement If the acknowledgement is for a checkpoint generated here, then all data and checkpoint tokens up to the corresponding checkpoint marker can be purged from *recoverylog*. Otherwise the acknowledgement is destined for an upstream *vertex*, and the sequence number in it is recorded for use by the application layer which defers relay of any acknowledgement until a copy of that acknowledgement has been received from each downstream adjacent *vertex*.

restart request Following a *flushed* token, the contents of the recovery log are returned in the sequence stored, including checkpoint markers, as if generated for the first time.

If this *end point* is the head of an in-coming edge, then the arriving token may be one of the other three types.

flushed This signals start of the recovery proper.

application data The token is enqueued in a holding buffer.

checkpoint marker If the checkpoint marker is destined for a downstream *vertex*, the *ID* of this *end point* is added to the path stored in the checkpoint marker before the latter is enqueued in the holding buffer. Otherwise it marks the end of a block of tokens which can be acknowledged, so the entries in the holding buffer are copied to the input queue and the checkpoint marker is sent as an acknowledgement to the tail of the *edge* associated with this *end point*. The acknowledgement will thence be relayed via the path stored in it to the source of the corresponding checkpoint marker.

The *end point receive()* operation is unchanged from figure 4. In figure 10, it is seen that arriving tokens are not copied into *inputq* until a checkpoint marker signalling end of block arrives. Thus, no tokens will be processed by the application code *vertex* until an acknowledgement has been sent for the corresponding checkpoint marker.

Figure 11 shows modifications made to the application code to exploit the fault-tolerance provision of the rollback-recovery protocol. At startup, a restart request is sent to each upstream adjacent

```
app() {
    for (each adjacent upstream vertex)
        endpoint[vertex]->send(restart); // for restart
    do {
        Token t = get_next(); // i.e. call receive() on
                              // whichever end point(s)
                              // to get next input token
        if (! t.checkpoint) {
            process(t);       // do any local processing
            put_any();        // output all result
                              // tokens by calling send
                              // on some end point(s)
        } else
            newest_cp[t.sender] = t;
        for (u = each upstream vertex) {
            if (done with tokens up to newest_cp[u]) {
                for (e = each outgoing endpoint)
                    e.send(newest_cp[u]);
            }
            Token ack = null; // compares as oldest
            for (v = each downstream vertex)
                ack = oldest(endpoint[v].newest_ack[u], ack);
            if (newer(ack, last_ack_sent[u])) {
                endpoint[u]->send(ack);
                last[u] = ack;
            }
        }
    } while (! finished_processing());
}
```

Fig. 11. Enhancing the application code to handle acknowledgements.

vertex. The essential processing of data tokens remains unchanged. However, the application code must in addition coordinate the forwarding of checkpoint tokens and relay of acknowledgements. This processing is clearly application specific, but typically not complex. For example, a simple filter can forward each checkpoint marker as it arrives. An aggregating filter would not forward any checkpoint marker until it has processed all input tokens. Figure 11 shows a rather general case where the latest checkpoint number for each upstream adjacent *vertex* and *acknowledgement* for each downstream adjacent *vertex* are maintained in two arrays and both checkpoint marker and acknowledgement processing is performed in the loop over upstream adjacent *vertices* at the bottom

of the main application loop. As described earlier, the code only relays an acknowledgement to an upstream adjacent *vertex* when all downstream adjacent *vertices* have acknowledged it.

C.2 Verification

The key properties of the rollback-recovery protocol when operated within a single *segment* of a *slice* of *thickness* 2 in a *uniform* graph are enumerated below. Clearly, multiple *segments* in a *slice* are quite independent as far as rollback recovery is concerned. In the context of this discussion, ***sources*** and ***sinks*** are those of the *segment*, and not necessarily those of the containing graph.

1. A token is only released for processing within the application layer of any ***sink*** when the checkpoint marker first following it has been received there, and an acknowledgement sent. The FIFO ordering of tokens within an *edge*, and the correct processing of checkpoint markers by application code, ensure that the checkpoint marker inserted by a ***source*** into a token stream to mark a *block* of tokens will arrive directly after (the subset of) those tokens at the ***sinks***.
2. All tokens are preserved in the ***sources*** until acknowledged as received by all ***sinks***.
3. Following from 1 and 2, if the *central vertex* fails, the ***sources*** are guaranteed to hold, in their recovery logs, all tokens which had been sent to the failed *vertex* but not acknowledged, or passed to the application layer, in the ***sinks***. There may be tokens in these logs which had been received by all ***sinks***, but for which the acknowledgement had not yet reached the ***sources***.
4. If the *central vertex* restarts, or fails and is replaced, then the application layer in that *vertex* requests restart from all ***sinks***.
5. Each incoming *end point* in the *central vertex* discards all tokens received after startup until receipt of the first token of type *flushed*; such tokens will have been in transit at failure.
6. In recovery, the new or restarted *vertex* reprocesses all tokens in any block not acknowledged by all ***sinks***.
7. Provided the processing which takes place in the application layer is deterministic, the new or replacement *vertex* will generate the same output tokens, as were generated before failure, for any work which it redoes during recovery; the ***sinks*** will be able to recognise a repeated block.
8. The ***sources*** and ***sinks*** can support failure of the *central vertex* even while recovery is in progress.
9. Since the *segments* of a *slice* of *thickness* 2 are independent the computation in a slice of *thickness* 2 is 1-fault-tolerant with regard to the *inner vertices* of that *slice*.

C.3 Rollback-Recovery in Overlapped *Slices* of *Thickness* 2

If a *digraph* is covered by overlapped *slices* of *thickness* 2 then clearly non-adjacent *slices* do not overlap each other. The bounding of the protocol within a *slice* ensures that concurrent failure of any single *inner vertex* within each of a pair of non-overlapping *slices* can be tolerated.

Where two *slices* overlap, checkpoint markers generated by the upstream *slice* do pass into the downstream *slice*, but are treated as data tokens by the *sources* of the downstream *slice*. Similarly, checkpoint markers generated by the downstream *slice* pass through the *sinks* of the upstream *slice* but are treated as data tokens there. When a *vertex* (re)joins a computation, the checkpoint markers generated by the *end points* of its outgoing *edges* will be recognised as more recent by *vertices* at distance 2 downstream due to the later timestamp. The *end points* of incoming *edges* must assume that no data has been received in order to avoid incorrectly rejecting tokens as duplicates. While

this guarantees that a single failure can be tolerated by any overlapped *slice*, it points to a limit on tolerance of concurrent or nearly concurrent failures.

In the example shown in figure 12, there are two overlapping *slices*, $d1 = \{v1, v2, v3\}$ and $d2 =$

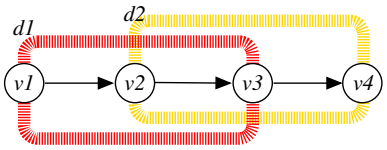


Fig. 12. Example graph showing overlap of two *slices*.

$\{v2, v3, v4\}$ each having a single *inner vertex*, $v2$ and $v3$ respectively. There are two cases.

1. *v3* fails and is replaced (by *v3'*), then *v2* fails and is replaced (by *v2'*).
2. *v2* fails and is replaced (by *v2'*), then *v3* fails and is replaced (by *v3'*).

In case 1, tolerance of failure of *v2* is supported immediately after recovery of *v3'*, because tolerance of failure of *v2* is not dependent on the recovery log in *v3*.

In case 2, there can be a period after *v2'* recovers during which its recovery log is not consistent with *v4*. Prior to its failure, *v2* was backing up tokens not yet acknowledged by *v4*. The tokens giving rise to the earliest of these may have already been acknowledged by *v3*, and thus discarded from *v1*. Such tokens will not be recreated during recovery of *v2'*. While failure of *v3* can eventually be tolerated following failure and recovery of *v2*, the size of the window during which it can't is determined by the nature of the application in *v3*. If this combines many input tokens in an aggregate result, then tolerating a single failure of *v2* could bar tolerance of a subsequent failure of *v3* for a significant time.

Accordingly, the only guarantee that may be given regarding overlapped *slices* is that the overall graph is tolerant of a single failure. On the other hand, the overhead is low.

V. Conclusions

It has been argued in this work 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 various styles of 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, by avoiding the need for any checkpoint of process state, but existing proposed protocols 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 class of *digraphs*, *uniform digraphs*, has been identified which, in having no alternate *walks* of differing length, suggest a convenient mechanism for implementing rollback-recovery. 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 token stream; these are returned as acknowledgements.

By way of example, the operation of a protocol is described for a *slice* of *thickness* 2. The protocol is shown to support 1-fault tolerance for the *inner vertices* of a *slice* and, by overlapping such *slices*, to tolerate at least a single fault in an arbitrary *uniform digraph*.

In addition to practical evaluation of the protocol, initially in the context of distributed query processing, 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. [25]

References

[1] David Abramson, Rok Sosic, Johnathan 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. IEEE Computer Society.

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

[3] Jean bacon, Ken Moody, John Bates, Richard hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *Computer*, 33(3):68–76, June 2000.

[4] Guruduth Banavar, Tushar Chandra, Bohdi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish subscribe systems. In *International Conference on Distributed Computing Systems*, pages 262–272, Austin, Texas, USA, May 1999. IEEE Computer Society.

[5] Adam Beguelin and Jack J. Dongarra. Graphical development tools for network-based concurrent supercomputing. In *Supercomputing*, pages 435–444, Albuquerque, New Mexico, United States, November 1991. ACM Press.

[6] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–54, December 1993.

[7] Kenneth P. Birman. A review of experiences with reliable multicast. *Software - Practice and Experience*, 29(9):741–774, July 1999.

[8] Wim Böhm, Walid Najjar, Bhanu 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.

[9] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Alexander Kreutz, Stefan Pröls, Stefan Seltzsam, and Konrad Stocker. Objectglobe: Ubiquitous query processing. *The VLDB Journal*, 10(1):48–71, August 2001.

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

[11] Tushak Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[12] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distribute stream processing. In *Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[13] Abhinandan Das, Johannes Gehrke, and Mark Riedewald. Approximate join processing over

data streams. In *International Conference on Management of Data*, San Diego, CA, USA, June 2003. ACM Press.

[14] Renzo Davoli, Luigi-Alberto Gianchini, Özalp Babaoglu, Alessandro Amoroso, and Lorenzo Alvisi. Parallel computing in networks of workstations with paralex. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):371–387, April 1996.

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

[16] Jack 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.

[17] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *International Symposium on Computer Architecture*, pages 126–132. ACM Press, December 1974.

[18] Roger E. Eggen and John 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.

[19] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[20] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of pulish subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[21] G. Fagg and Jack 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, Lake Balaton, Hungary, 2000. Springer-Verlag.

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

[23] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Fransisco, 1999.

[24] Jean-Luc 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.

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

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

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

[28] Gilles 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.

[29] Jaewoo Kang, Jeffrey F. Naughton, and Stratis D. Viglas. Evaluating window joins over unbounded streams. In *International Conference on Very Large Databases*, Hong Kong, China, August 2002. Morgan Kaufmann.

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

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

[32] Michael J. Litzkow, Miron Livny, and Matt 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.

[33] Walid Najjar and Jean-Luc Gaudiot. A data-driven execution paradigm for distributed fault-tolerance. In *SIGOPS European Workshop*, pages 1–5, Bologna, Italy, 1990. ACM Press.

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

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

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

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

[38] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A.A. Fernandes, and Rizos Sakalleriou. Distributed query processing on the grid. In *International Workshop on Grid Computing*, pages 279–290, Baltimore, MD, USA, November 2002. Springer Verlag.

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

[40] Paul Stelling, Cheryl DeMatteis, Ian T. Foster, Carl Kesselman, Craig A. Lee, and Gregor von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.

[41] Andrew S. Tannenbaum, editor. *Computer Networks*. Prentice-Hall, 1991.

[42] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Anthony Hey, and Geffrey Fox, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 299–335. John Wiley and sons Ltd., 2003.

[43] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Uğur Cetintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *Bulletin of the Technical Committe on Data Engineering*, 26(1):3–10, March 2003.

[44] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *International Conference on Very Large Databases*, Hong Kong, China, August 2002. Morgan Kaufmann.