Scalability of the Microsoft Cluster Service

Werner Vogels, Dan Dumitriu, Ashutosh Agrawal, Teck Chia, Katherine Guo

Department of Computer Science, Cornell University[†]

vogels@cs.cornell.edu

Abstract

An important argument for the introduction of software managed clusters is that of scale: By constructing the cluster out of commodity compute elements, one can, by simply adding new elements, improve the reliability of the overall system in terms of performance and in availability. The limits to how far such a cluster can be scaled seems to be dependent on the scalability of its management software, which in its core has a collection of distributed algorithms to guarantee the correct operation of the cluster. The complexity of these algorithms makes them a vulnerable component of the system in terms of their impact on the overall scalability of the system.

This paper examines two of the distributed components of the Microsoft Cluster Service [8] that are most likely to have an impact on its scalability: the membership and the global update managers. The first sections of the paper will provide some general background on these distributed services and scalability issues. After that the algorithms used to implement these service are described in detail and an analysis of their impact on scalability is given. The scalability analysis is based on an off-line analysis of the algorithms as well as the results of on-line experiments on a cluster with a, in MSCS terms, large number of nodes.

1 Distributed Management

In distributed management software two components are considered basic building blocks: a consistent view about which nodes are on-line, and the ability to communicate with these nodes in an all-or-nothing fashion [2].

The first building block is captured in a *membership* service: all nodes participate in a consensus algorithm to agree on the current set of nodes that are up and running. The system makes use of a failure detection mechanism that monitors heartbeat signals or actively polls other nodes in the system. The failure detector will signal the membership service whenever it suspects the failure of a node in the system. The membership service will react to this by triggering the execution of a distributed algorithm at all the nodes in the system, in which they agree upon which nodes have failed and which are still available. The joining of a new member in the system, does not require the nodes to run the agreement protocol, but can often be handled through a simple update mechanism.

The second fundamental component provides a special communication facility, with guarantees that exceed the properties provided by regular communication systems. Often in the process of managing a distributed system it is necessary to provide the same information to a set of nodes in the system. We can simplify the software design of many of the components on the receiving side of this information if we can guarantee that if one node receives this information, that all nodes will receive it. This *atomicity* guarantee allows nodes to act immediately upon reception of a message, without the need for additional synchronization. Often this atomicity guarantee is not sufficient for a system, as it does not only need be assured that all nodes will receive the update, but that all nodes will see the updates in the same order. This *total order* property makes the communication module a very powerful mechanism in the control of the distributed operation of the distributed system.

2 Practical Scalability

This paper examines the membership and communication services of the Microsoft Cluster Service (MSCS) with an emphasis on their impact on the scalability of the system. MSCS, as shipped, officially supports 2 nodes, but in reality the software can be run on a 16-node NT server cluster. At Cornell the software is extended to run on 32 nodes and a research project is underway to make the system scale to larger numbers.

Making systems scale in practice centers around the use of mechanisms to reduce the dependency of the algorithms on the number of nodes. In the past two approaches have been successful in finding solutions to problems of scale: The first is to reduce the synchronous behavior of the system by designing messaging systems and protocols that allow high levels of concurrent operation, for example by de-coupling the

[†] The reliable cluster computing research of the Reliable Distributed Computing Group at the department of Computer Science at Cornell University is supported by DARPA/ONR under contract N0014-96-1-10014 and by Intel Corporation and Microsoft Corporation.

sending of messages from the collecting of acknowledgements. The second approach is to reduce the overall complexity of the system. By building the system out of smaller (semi-)autonomous units and connecting these units through hierarchical methods, growing the overall system has no impact on the mechanism and protocols used to make the smaller units function correctly.

A third, more radical approach, which is under development at Cornell, makes use of gossip based dissemination algorithms. These techniques significantly reduce the number of messages and the amount of processing needed to reach a similar level of information sharing among the cluster nodes.

Given that cluster systems such as MSCS are used for enterprise computing, any instability of the system can have severe economic results. There is a continuous tradeoff between responsive failure handling and the cost of an erroneous suspicion. The system needs to detect and respond to failures in a very timely matter, but designers may choose a more conservative approach given the significant cost of an unnecessary reconfiguration of the system, caused by an incorrect failure suspicion. In general cluster server systems run compute and memory intensive enterprise applications and these systems experience a significant load at times, reducing the overall responsiveness. Scaling failure detection needs intelligent mechanisms for fault investigation [6,11] and requires the failure detectors to be able to learn and adapt to changes [7].

3 Scalability goals of MSCS

The Microsoft Cluster Service is designed to support two nodes, with a potential to scale to more nodes, but in a very limited way. MSCS successfully addresses the needs of these smaller clusters. The cluster management tools are a significantly improvement over the current practice and they are a major contribution to the usability of clusters overall.

The research reported here is concerned with scaling MSCS to larger numbers of nodes (16 - 64, or higher), which is outside of the scope of the initial MSCS design. There are three areas of interest:

1. Can the currently used distributed algorithms be a solid foundation for scalable clusters?
2. Are there any architectural bottlenecks that should be addressed if MSCS needs to be scalable?
3. If MSCS is extended with development support for cluster aware applications are the current distributed services a good basis for these tools?

This paper should not be seen as criticism of the current MSCS design. Within the goals set for MSCS it functions correctly and will scale to numbers larger then originally targeted by the cluster design team.

4 Cluster Management

The algorithms used in MSCS for membership and total ordered messaging are a direct derivative of those developed in the early eighties for Tandem as used in the NonStop systems [3,4]. Nodes in a Tandem system communicated via pairs of proprietary inter-processor busses, which, in 1985, provided a 100 Mbit/second transfer rate. Parts of the messaging side of the algorithms was implemented in interrupt handlers to provide minimal system overhead.

Although MSCS has a kernel module that implements some of the messaging and failure detection, the membership and global update algorithms are implemented in an NT service, the *Cluster Service*, which runs at user level. The Cluster Service holds in total 11 managers, each responsible for a different part of the cluster service functionality. Next to the membership and communication managers, there are managers for resource and failover management, for logging and checkpointing, and for configuration and network management.

In the following sections three of the components are examined in detail: first the kernel module which holds the cluster communication and failure detection functionality. Secondly the join process and the failure reconfiguration of the membership module are analyzed. The last analysis is that of the global update communication module.

5 Cluster Network

MSCS provides a kernel based cluster network interface, *ClusNet*, which presents a uniform interface to networks available for intra-cluster communication. ClusNet supports basic datagram communication to each of the nodes, using an addressing scheme based on simple node identifiers which are assigned to nodes when they are first configured for use in the cluster. To support reliable communication ClusNet provides a transport interface used by MS-RPC.

ClusNet is capable of managing a redundant networking infrastructure, automatically adapting packet routing in case of network failure.

5.1 Node Failure Detection

MSCS implements its Failure Detection (FD) mechanism using heartbeats. Periodically every node sends a sequenced message to every other node in the cluster, over the networks that are marked for internal communication. Whenever a node detects a number of consecutive missing heartbeats from another node it sends an event to the cluster service which uses this event to activate the membership reconfiguration module.

In the current MSCS configuration heartbeats are sent every 1.2 seconds and the detection period for a node suspicion is 7.2 seconds (6 missed heartbeats). The timing values are not adaptive.

The cluster network module does not exploit any broadcast or multicast functionality, and thus each heartbeat results in (*number_of_nodes-1*) point-to-point datagrams. In our test setup of 32 nodes, the cluster background traffic related to heartbeats is 800 messages per second. With 32 nodes active and an otherwise idle network the mechanism works flawless and the packet loss observed was minimal. Tests which replaced the Fast-Ethernet switches with hubs showed that the packet trains sometimes caused significant Ethernet-level collisions on the shared medium. Adding processing load to the systems resulted in variations in the inter-transmission periods. False suspicions were never seen.

When adding processing load and additional load on the network frequently single heartbeat misses where observed, but the values for generating a failure suspicion event so conservative that never any false suspicions were generated.

6 Node Membership

The MSCS membership manager is designed into two separate functional modules: the first handles the joining of nodes and a second, *regroup*, implements the consensus algorithm that runs in case of a node failure.

6.1 Join

The join algorithm starts with a discovery phase in which the joining node attempts to find other nodes in the cluster. If this fails the node tries to form a cluster by itself, the details of the *cluster-form* operation are outside of the scope of the paper. After the node has discovered which cluster nodes are currently running it selects one of the nodes and petitions for membership of the cluster. The selected node, dubbed the *sponsor*, announces the new node to all active cluster members, transfers all the up-to-date cluster configuration to the new node, and waits for the node to become active. The different phases of the join and their distributed complexity are described in detail in the following paragraphs

Phase 1: **Discovery**. When a cluster service starts is attempts to connect to each of the other known nodes in the cluster, using RPC over a regular UDP transport. This sponsor discovery mechanism has a high degree of concurrency: a thread is started for each connection probe. The joiner waits for all threads to terminate, which occurs after the RPC binding operation fails after

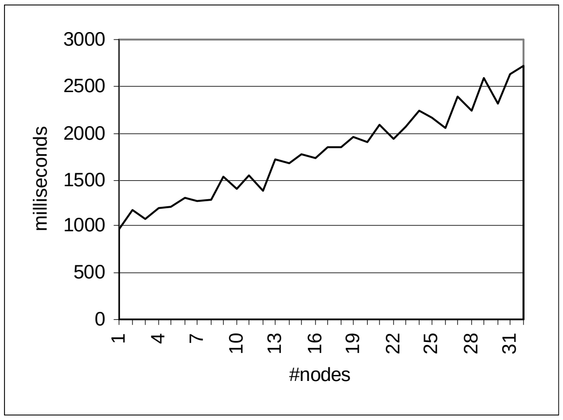

Figure 1. Join latency under ideal conditions

a time-out or when a connection is established. As the joiner waits for all threads to terminate, the delay the joining node experiences is based on the time-out period of an RPC connection to a single node that is not up. The timeout value for RPC out-of-the-box is approximately 30 seconds, but it can be manipulated to reduce the discovery phase to 10 seconds.

In all observed cases, the joining node always selected the holder of the cluster IP address to sponsor its join. The cluster IP address is a single address that migrates to a node that functions as the access point for administrative purposes: if the cluster is running there is *always* a node that holds this IP address. By modifying the startup phase to start by attempting to connect to this address first before probing all the other nodes, it is possible to reduce this phase of the join process to under a second. This approach also avoids starting a number of threads that is equal to the number of nodes in the cluster.

Phase 2: **Lock.** From the nodes that are up, the joiner selects one node to *sponsor* its membership in the cluster. The first action by the sponsor is to acquire a distributed global lock to ensure that only a single join is in progress. Acquiring of the lock is performed using a global update (GUP) method.

The use of GUP makes this phase is dependent on the number of active nodes. Details on the performance and scalability of GUP can be found in section 7.

Phase 3: **Enable Network**: Using a sequence of 5 RPC calls to the sponsor the joiner retrieves all information on current nodes, networks and network interfaces. Following this the joiner performs an RPC to each active node in the cluster for each interface a node is listening on, and the contacted node in return performs an RPC to the joiner to enable symmetric network channels. After this sequence the node security contexts are established which again requires the joining node to contact all other active nodes in the cluster, in sequence.

This phase depends on the number of active nodes in the cluster. An unloaded 31 nodes cluster, on average, performs this sequence of RPC's in 2-4 seconds. On a moderately loaded cluster, frequently this phase takes longer then 60 seconds, causing the join operation to time-out at the sponsor, resulting in an abort of the join.

Phase 4: **Petition for Membership:** The joiner requests the sponsor to insert the node into the membership. This is a 5-step process directed by the sponsor.

1. The *sponsor* broadcasts to all current members the identification the joining node.
2. The sponsor sends the membership algorithm configuration data to the joiner
3. The sponsor waits for the first heartbeat from the new joiner.
4. The sponsor broadcasts to all current members that the node is alive
5. The sponsor notifies the joiner that it is inserted in the membership

The broadcasts are implemented as series of RPC calls, one to each active node in the cluster. On an unloaded cluster and network the serialized invocation of RPC to 30 nodes takes between 100 and 150 milliseconds. When loading the systems with compute and IO tasks, the RPC times vary widely from 3 millisecond to 3 second per RPC. Broadcast rounds to all 30 nodes were observed taking more then 20 seconds to complete (with exceptions up to 1 minute). As this phase is under control of the sponsor the join is not aborted because of a time-out. It can abort on a communication failure with any of the nodes.

In step 3 the detection of the new heartbeat is delegated to ClusNet, which performs checks every 600 millisecond, resulting in an average waiting period between 0.6 and 1.2 seconds

Phase 5: **Database synchronization.** The joiner synchronizes its configuration database with the sponsor. In the experimental setup this database was of minimal size and never out-of-date. As the retrieving of the database updates is not depended on cluster size, not further tests were performed in this phase.

Phase 6: **Unlock**. The newly joined node uses its access to the global update mechanism to broadcast to all nodes that it now is full operation and that the global lock should be released.

The join operation is very much dependent on the number of nodes in the system. Figure 1 show the times for a join under optimal conditions. All RPC calls in the algorithms are serialized and at minimum there are *(10 + 7 * number_of_nodes)* calls. Joining the 32nd node to the cluster requires at least 227 sequential RPC's. This approach collapses under load, frequently it is impossible to join any nodes if only a moderate load is placed on the nodes and the system has more then 10-12 nodes.

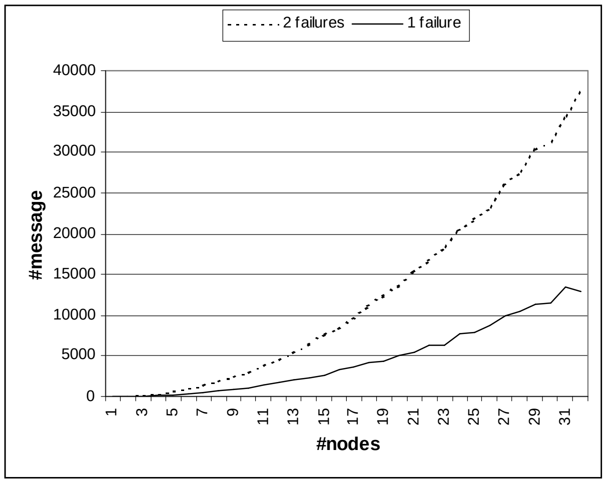

Figure 2. Number of messages in the system during regroup

6.2 Regroup

Upon the receipt of a node failure event generated by ClusNet the Cluster Service starts the reconfiguration algorithm, dubbed *regroup*. The algorithm runs in 5 phases, with the transition to each new phase determined after its is believed that all other nodes have finished this phase, or when, in the first two phases, timers expire.

During regroup the nodes periodically (300ms) broadcast their current state information to all other nodes using unreliable datagrams. The state is a collection of bitmasks, one for each phase, describing whether a node has indicated it has passed a phase. It is not necessary for each node to have heard for each other node in a phase; information about which other nodes a certain node has heard of is shared. For example if node 1 indicates that it has received a regroup message from node 2, node 3 uses this without that it actually needs to receive a message from node 2 in that phase. Also included in the state is a connectivity matrix in which nodes record whether they have seen messages from the other nodes and what connectivity information has been recorded by the other nodes.

The 5 phases of the regroup algorithm are the following:

Phase 1: **Activate**. Each node waits for a local clock tick to occur so that it knows that its timeout system can be trusted. After that it starts sending and collecting status messages. It advances to the next stage if

1. All current members have been detected to be active (e.g. there was a false suspicion),
2. If there is one single failure and a minimal time-out has passed or,

3. When the maximum waiting time has elapsed and several members have not yet responded.

The minimum timeout for phase 1 is 2.4 second, if all but one node have responded in this time period it is assumed that there was a single failure and the algorithm moves to the next phase. If multiple nodes do not respond, the algorithm waits for 9.6 seconds to move to the next phase. If for some reason the regroup algorithm times out in a different phase or when there are cascading starts of the regroup algorithm at several nodes, the algorithm executes in *cautious* mode and always waits for the maximum timeout to expire.

Phase 2: **Closing**. This stage determines whether partitions exist and whether the current node is in a partition that should survive. The rules for surviving are:

1. The current membership contains more than half the original membership.
2. Or, the current membership has exactly half the original members, and there are at least two members in the current membership and this membership contains the tie breaker node that was selected when the cluster was formed.
3. Or, the original membership contained exactly two members and the new membership only has one member and this node has access to the quorum resource.

After this the new members select a tie breaker node to use in the next regroup execution. This tiebreaker then checks the connectivity information to ensure that the surviving group is fully connected. If not it prunes those members that do not have full connectivity. It records this pruning information in its regroup state, which is broadcast to all other nodes. All move to stage 3 upon receipt of this information.

In case of incomplete connectivity information the tiebreaker waits for an additional second to allow all nodes to respond.

Phase 3: **Pruning**. All nodes that have been pruned because of lack of connectivity halt in this phase. All others move forward to the first cleanup phase once they have detected that all nodes have received the pruning decision (e.g. they are in phase 3).

Phase 4: **Cleanup Phase One**. All surviving nodes install the new membership, mark the nodes that did not survive the membership change as down, and inform the cluster network to filter out messages from these nodes. Each node's Event Manger then invokes local callback handlers to notify other managers of the failure of nodes.

Phase 5: **Phase Two**. Once all members have indicated that the Cleanup Phase One has been successfully executed, a second cleanup callback is invoked to allow a coordinated two-phase cleanup. Once all members have signaled the completion of this last cleanup phase they move to the regular operational state and seize the sending of regroup state messages.

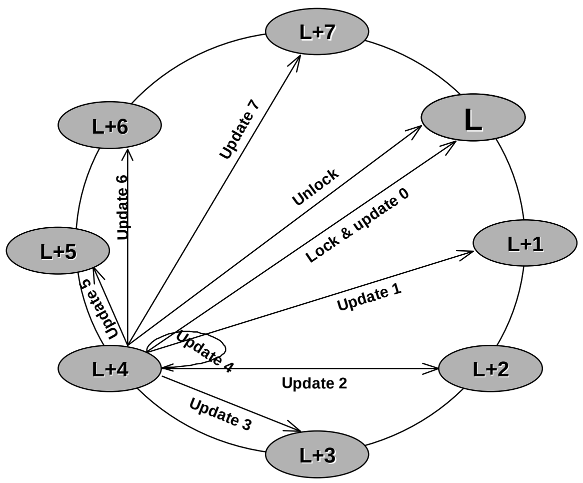

Figure 3. Global Update Sequence

The regroup algorithm in its first two phases is timer driven and the algorithm makes progress independent of the number of nodes in the cluster. The transitions of the next 3 phases are dependent on the number of nodes in the system, but the "information sharing" mechanism makes the system robust in dealing with sporadic message loss.

The state information is broadcast by sending point-to-point datagrams to each node in the cluster. With an inter-transmission period of 300 millisecond, and 31 nodes in the cluster, this generates a background traffic of over 3000 messages/second. A single failure reconfiguration has an average runtime of 3 seconds and thus generates around 10,000 messages. A two-node failure, with a full running cluster is likely to generate between 30,000 and 40,000 messages. Figure 2 details the observed messages in the system during regroup.

7 Global Update Protocol

It is essential for a distributed management system to have access to a primitive that allows consistent state updates at all nodes. MSCS uses the Global Update Protocol (GUP) for this purpose. Although the protocol is described as providing atomicity, its implementation has the stronger property of providing total ordering to its update messages.

When a node starts an global update operation, it first competes for a transmission lock managed by a node that is assigned the functionality of the *locker* node. Only one transmission can be in progress at a time. If the sender can not obtain the lock it is queued on the lock waiting list and blocks until it reaches the head of

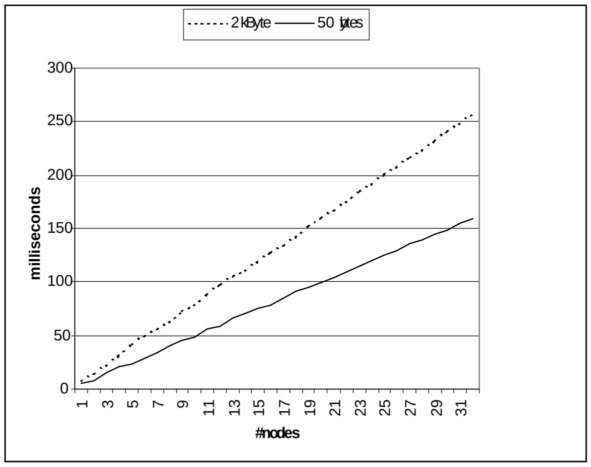

Figure 4. latency of GLUP under ideal conditions

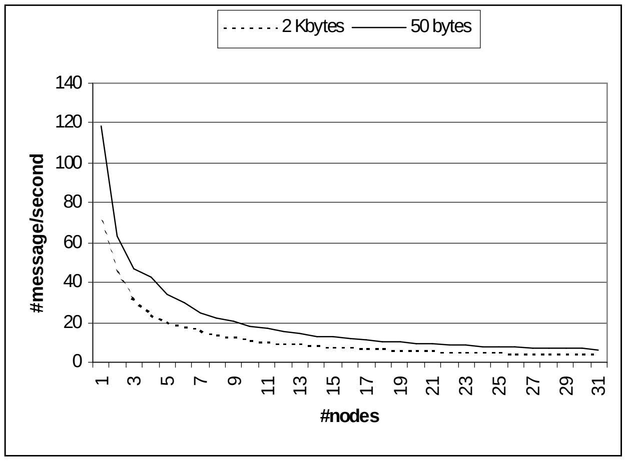

Figure 5. GLUP throughput under ideal conditions

the queue. With the lock request the sender also transmits its update information to the locker node which applies it locally, and stores the message for later replay under certain failure scenarios. While holding the lock the sender transmits its update to all other active nodes in the cluster and terminates the transmission with a final message to the locker node which releases the lock (see figure 3).

To transmit the messages to all other nodes, the sender organizes the cluster nodes into a circular list, ordered by NodeId. After it acquired the lock the sender send its updates starting with the node that is after the locker node in the list. The sender works through the list in order, wrapping when it reaches the last node in the cluster to the first node and stops when it once again reaches the locker node. The transmission is finished with an unlock message to the locker node.

Acquiring the lock before performing the updates guarantees that only one update is in progress at a given time, which gives the protocol the total ordering property. Atomicity (if one surviving node applies the update, all other surviving nodes will) is achieved through the implementation of a number of fault-handling scenarios.

1. *The sender fails*: the locker node takes over the transmission and completes it.
2. *A receiver fails*: wait for the regroup to finish and then finish the transmission.
3. *The locker node fails*: the next node in the node list is assigned locker functionality and the sender treats it as such.
4. *The sender and locker fail*: if the node following the locker has received the update already, in its role as new locker it takes over the transmission.
5. *All nodes that received an update and the sender fail*: pretend the update never happened.

The protocol is implemented as a series of RPC invocations. If an RPC fails, the sender waits for the regroup algorithm to run and install a new membership. GUP will then finish the update series based on the new membership.

Given the strict serial execution of the protocol, its performance is strongly dependent on the number of nodes in the system. The implementation enforces no time bound on the execution of an RPC and any node can introduce unbounded delays as long as RPC keep-alives are being honored.

Repeated measurements show huge variations in results, with the variations being amplified as the number of nodes increases. When a moderate load is placed on the nodes it becomes impossible to produce stable results. These variations can be contributed to the RPC trains, which repeatedly transfer control to the operating system while blocking for the reply. Upon arrival of the reply at the OS level, the Cluster Service needs to compete with other applications that are engaged in IO, to regain CPU control. The non-determinism of the current load state of the system introduces the variances.

The latency of the protocol in an ideal setting is shown in figure 4, the message throughput in figure 5. With 32 nodes the system can handle 6 small (50 bytes) updates/second or 4 larger (2 Kbytes) updates/second.

With systems under a load the protocol breaks down with more then 12 nodes in the cluster. With 10 nodes frequently transmissions are observed that take 2-5 seconds to complete. With 32 nodes transmission times up to one minute were recorded.

8 Discussion

When evaluating the scalability of the distributed components of MSCS it is necessary to separate two issues: the algorithms used and their particular implementation.

8.1 Failure Detection

MSCS is willing to tolerate a long period of silence (7 seconds) before a failure suspicion is raised. This allows for the implementation of mechanisms that can easily deal with large number of nodes. The important scale factor is the number of messages that the nodes need to process both at the sending and the receiving side. Implementing the heartbeat broadcast using repeated point-to-point datagrams does not introduce any problems with 32 nodes, but there is a clear processing penalty at the sender and it will limit the growth to larger numbers.

In an unstructured heartbeat scheme (every node sends heartbeats to all other nodes), the load on the sender and on the network can be significantly reduced by using a true multicast primitive for disseminating the heartbeats. It also removes the sender's dependency on the number of nodes in the system. However, the number of messages a receiver has to process remains proportional to the number of nodes in the system.

More structured approaches have been proposed to reduce the overall complexity of failure detection by imposing a certain structure on the cluster, and localizing failure detection within that structure. A popular approach is to organize the cluster nodes in a logical ring [1,5] where nodes only monitor neighbor nodes in the ring and a token rotates through the ring to disseminate status information. In this scheme however, the token rotation time is dependent on the number of nodes, and the scheme thus has clear scalability limits.

Another aspect of scaling failure detection is the increased chance of multiple concurrent node failures in the cluster. The MSCS mechanism handles multiple failures just as efficient as single failures, while most of the structured failure detection schemes have problems with timely detection of multiple failures and fast reconfiguration of the imposed structure.

Currently the most promising work on failure detection for larger systems is the use of gossip and other epidemic techniques to disseminate availability information [6]. These detectors monitor hundred's of nodes while still providing timely detection, without imposing any significant increased load on nodes and networks.

8.2 Membership Join

The observation that it frequently was impossible to join the 15th or higher node into the cluster is an artifact of the fact that MSCS was not implemented with a large number of nodes in mind. The join reject happens in the phase that is not under control of the sponsor node and where the new node is setting up a mesh of RPC bindings and security contexts with all other active nodes. With 32 nodes this phase is close to a 100 RPC's and any load on the nodes causes significant variations in these serialized executions.

There is no fundamental solution to the problem; if the RPC infrastructure needs to be maintained, the setup phase is needed and some tolerance is needed to allow the mesh to be established. A possible solution would for the joiner to update the sponsor on its progress in this phase to avoid a join rejection.

8.3 Membership Regroup

The membership reconfiguration algorithm works correct under all tested circumstances, independent of the number of nodes used. There are two mechanisms that ensure that the operation performs well, even with a larger number of nodes: (1) The operation is fully distributed, the constant broadcasting of state allows node to rely solely on local observation of global state. (2) The sharing of "*I-have-heard-from-node-X*" information among nodes, makes that the nodes can move to the next phase without having received status messages from all nodes.

Given that a node failure suspicion is not raised until 7 seconds of silence by a node and the first phase of regroup waits for an additional 3 seconds, a problematic node has 10 seconds to recover from some transient failure state. As no false suspicions were ever observed, the timeouts in the first two phases of regroup can be considered to be very conservative. In all observed cases the current membership state was already established well within a second, the remaining time (2-9 seconds) was spent waiting for the failed nodes to respond. As the first phase is dominant in the execution time of the whole regroup operation, a reduction in time can be achieved by combining the failure detection information with the observed regroup state.

A major concern in scaling the regroup operation is the number of messages exchanged. A typical run with 32 nodes generates between 10,000 and 40,000 messages. The status message broadcasts are implemented as series of point-to-point datagrams, which has two major effects: (1) the number of messages generated for the regroup operation grows exponential with the number of nodes and (2) the transmission of 32 identical messages every 300 milliseconds introduces a significant processing overhead at the sender. The regroup algorithm is run at the cluster service, which introduces a user-space/kernel transition for each message, with associated overhead. Introduction of a multicast primitive will allow the implementation to scale at least linearly with the number of nodes and would remove the processing over from the sender of status messages.

8.4 Global Update Protocol

The absence of any concurrency in the message transmission in GUP causes a strict linear increase in latency and decrease in throughput when the number of nodes in the cluster grows.

This serialized and synchronous nature of the protocol is amplified in the particular MSCS implementation. The protocol was originally developed for updating shared OS data-structures, with the update routines running in device interrupt handlers. In MSCS the protocol is implemented uses a series of RPC calls to user-level services. This change in execution environment exposes the vulnerability of the strict serialized operation.

There is no quick solution for the problems that this GUP implementation presents us with. To emulate the original Tandem execution environment the Cluster Service would need to be implemented as a kernel service, which at this point seems impractical.

Replacing GUP with a protocol that provides the same properties but exhibits a more scalable execution style seems preferable. This introduces a number of other complexities, for example many of the currently popular total ordering protocols rely on a tight integration of membership and communication to ensure correct failure handling. This would result in replacing *regroup* as well as GUP.

9 Conclusions

In this paper some of the scalability aspects of the Microsoft Cluster Service were examined. When revisiting the three questions from section 3 the following is concluded:

Can the currently used distributed algorithms be a solid foundation for scalable clusters?

Both failure detection and *regroup* scale well to the numbers that were tested in this paper. When scaling to larger numbers the state processing at receivers will become an issue. The serialized nature of GUP limits its scalability to 10-16 nodes in the current MSCS setup.

Are there any architectural bottlenecks that should be addressed if MSCS needs to be scalable?

The major issue in both failure detection and *regroup* is the implementation of a broadcast facility using repeated point-to-point messages. This introduces a significant overhead on the sender and on the network, and needs to be replaced by a simple multicast primitive. The RPC trains in the membership join operation and in GUP, create a major obstacle for scalability, especially when the systems operate under a significant load.

If MSCS is extended with development support for cluster aware applications are the current distributed services a good basis for these tools?

Support for cluster aware applications has strong requirements in the area of application and component management and failure handling, and requires efficient communication and coordination services. These services would need to be implemented using GUP, which is, in its current form, unsuitable to provide such a service.

To support cluster aware applications a better integration of membership and communication is needed. This will allow for the implementation of a very efficient communication service with properties similar to GUP. Such a service is capable of providing a solid basis for application and component level management and failure handling, and will offer efficient communication and coordination services.

10 Future Work

Research is underway at the Cornell's Reliable Distributed Systems group to investigate and implement alternatives to the distributed management and networking modules in MSCS. Goal is to allow the system to perform well under the scenarios tested for this analysis and to scale to larger numbers (256 nodes and above) at reasonable cost. Recent results such as the scalable failure detection [6] are very promising and show that managing these numbers of nodes is feasible.

In a related project, dubbed Quintet [9,10], new tools are developed to construct highly available, cluster aware application servers. Quintet exploits MSCS features where possible, but at this point provides its own membership and communication modules.

Acknowledgements

Discussions with Jim Gray, Catharine van Ingen, Rod Gamache and Mike Massa have helped to shape the research reported in this paper. The advice of shepherd Ed Lazowska was very much appreciated. Thorsten von Eicken, S. Keshav and Brian Smith graciously contributed hardware to the world's largest *wolfpack* cluster.

References

[1] Badovinatz, P., Chandra, T.D., Gopal, A., Jurgensen, D., Kirby, T., Krishnamur, S., and Pershing, J., "GroupServices: infrastructure for highly available, clustered computing", unpublished document, December 1997

[2] Birman, K.P., *Building Secure and Reliable Network Applications*. Manning Publishing Company, and Prentice Hall, 1997

[3] Carr, R.,"Tandem Global Update Protocol", *Tandem Systems Review*, V1.2 1985.

[4] Katzman., J.A., et.al., "A Fault-tolerant multiprocessor system", United States Patent 4,817,091, March 28, 1989.

[5] Moser, L., Melliar-Smith, M., D. A. Agarwal, D., Budhia, R., and Lingley-Papadopoulos, C., “Totem A Fault-Tolerant Multicast Group Communication System”, *Communications of the ACM,* April 1996.

[6] Renesse, R. van, Yaron Minsky, Y., and Hayden, M., “A Gossip-Based Failure Detection Service”, in *Proceedings. of Middleware '98*, Lancaster, England, September 1998.

[7] Renesse, R. van, Birman, K., Hayden, M., Vaysburd, A., and Karr, D., “Building Adaptive Systems Using Ensemble”, *Software--Practice and Experience,* August 1998.

[8] Vogels, W., Dumitriu, D., Birman, K. Gamache, R., Short, R., Vert, J., Massa, M., Barrera, J., and Gray, J., "The Design and Architecture of the Microsoft Cluster Service -- A Practical Approach to High-Availability and Scalability", *Proceedings of the 28th symposium on Fault-Tolerant Computing,* Munich, Germany, June 1998.

[9] Vogels, W., Dumitriu, D., Panitz, M., Chipalowsky, K., Pettis, J., "Quintet, Tools for Reliable Enterprise Computing", submitted for publication, June 1998.

[10] Vogels, W., van Renesse, R., and Birman, K., "Six Misconceptions about Reliable Distributed Computing", *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998

[11] Vogels, W, "World Wide Failures", *Proceeding of the 1996 ACM SIGOPS Workshop,* Ireland 1996.