Chapter 5: High level design of a fault recovery protocol

5.1. Introduction

The last two chapters presented a large number of decisions regarding the design of mechanisms to perform fault recovery in a realtime network. In this chapter, we will summarize this information and place it into the context of a high level description of a fault recovery protocol, intended to work with the Tenet Suite enhanced with the DCM protocol (see Figure 2.2). This protocol will be called the Real Time Control Message Protocol (RTCMP), after the Internet Control Message Protocol (ICMP), the equivalent protocol in the Internet protocol stack.

The support for routing required by the fault recovery process may be implemented by modifying the existing routing software of the realtime network. A routing function is included in the software implementing the combined functionality of RCAP and DCM. We will use the term "RCAP+DCM" loosely to refer to either the combined protocols or to the software implementing them. Thus, the RTCMP implementation involves modifying the routing function of RCAP+DCM, and extending the control interface to allow RTCMP to use it appropriately. In this chapter, we will describe the interface and the functionality required from the routing element. The changes in software described will be relative to the routing software as currently implemented in RCAP+DCM. However, this model should also be applicable to a separate routing module implementation such as in the Tenet Suite 2 (36).

The rest of the fault recovery process will be implemented as a separate protocol, namely, RTCMP, which will run on all the nodes where the Tenet control protocols run. The RTCMP entity on a given node would exchange messages with neighboring RTCMP entities, the routing module, and the channel management process on the same machine. The other events which would trigger action from the RTCMP protocol are timer events and fault events generated by the network hardware. The interactions of the RTCMP protocol with the rest of the world is shown in Figure 5.1. Since this high level description is not

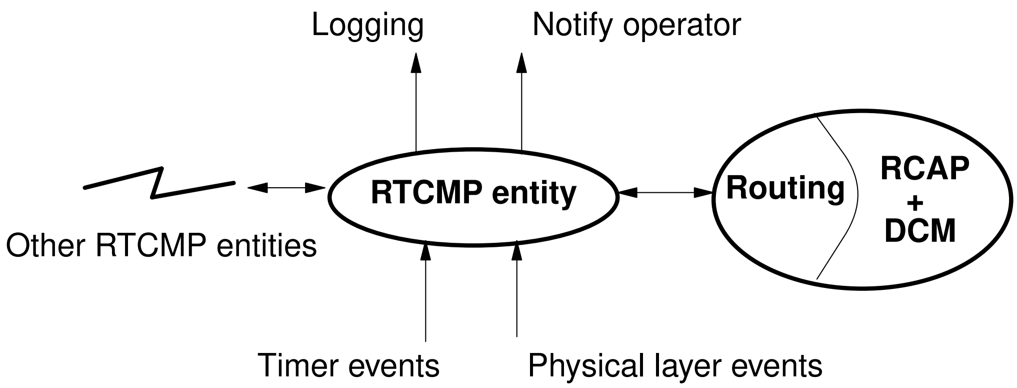

Figure 5.1: Interactions for RTCMP

targeted to a specific network or operating system, no implementation details will be provided.

5.2. Summary of design decisions

Starting from a completely general framework for realtime channel recovery in the beginning of Chapter 3, we narrowed down the possible choices, to the point where we can now describe the recovery process at a high level. In the previous two chapters, we made several design decisions, which we will summarize here.

The first set of decisions made were at a general level. We argued that a routing strategy that optimizes the network routes on a global level is too time consuming and unscalable, given the dynamics and scale of realtime networks. We chose a greedy strategy, in keeping with the strategy followed by the establishment protocol of the real-time network, where each channel would be routed in isolation, minimizing resource usage in this isolated context by using "min-hop" routing. In the context of this decision, the question of whether the fault recovery algorithm should be centralized or distributed became easier to answer, since distributed algorithms can implement greedy strategies equally well, and provide the benefits of scalability and robustness. We also chose to dynamically compute the routes from scratch for each failure, rather than pre-compute the configurations for all possible fault conditions, because of the dynamics of realtime channel establishment and teardown, and the space complexity associated with attempting to account for all possible configurations that a realtime network system could be in. We decided to further explore, in the course of our experiments, the question of the degree of information exchange during a distributed recovery.

Thereafter, we eliminated some of the variables in the experiments, such as the method used for failure detection, the route update process, the graph theoretical algorithm used to find the "min-hop" path, and the channel management process. We did this either by making simplifying assumptions in our simulator, or by assuming support from existing protocols. The remaining questions, open to investigation through simulation, were: the constraints to be used on the route selection process, the timing of the reroute and establishment attempts, the number and timing of retries, and the level of state prediction used to improve the recovery success. The issue of information exchange is implicit in the timing question and in the state prediction question, since the route update protocol exchanges information in the background continuously, so that, by waiting for the appropriate interval, the information needed is obtained without invoking an active request-response protocol.

The experiments conducted in Chapter 4 led us to the conclusion that Global locus of reroute, Immediate timing, Delayed retries, and Local prediction would provide a recovery success comparable to that of the schemes with maximum overhead for information exchange, but considerably faster. Methods that make decisions using the available information, rather than waiting for information to come in, seem to work best. The only case where waiting for information to come in seems to help is when an establishment request has failed, and the protocol is attempting a retry. The delay appropriate for the retry is close to the average round-trip latency of the network.

Fortunately, these choices are among the simplest that we looked at, which simplifies the task of designing a protocol based on them. For example, Local locus of rerouting is more complicated to implement that Global, since it involves a route computation model significantly different from the model described in Chapter 2. Global locus of reroute uses the same route computation model as RCAP+DCM, i.e., the route is computed at the source, and the shortest path which meets the delay constraint is used. Thus, the task of implementing this locus of reroute is the simplest, at least when the DCM protocol is present. Immediate timing is also the simplest to implement, since it does not require any timing cooperation between nodes. Similarly, implementing Global state prediction would have involved significant changes to the route computation software of RCAP+DCM. Instead, it turns out that Local prediction performs adequately. Local prediction does involve some minimal changes to the current RCAP+DCM routing software and its interface; these changes will be described in this chapter.

5.3. Description of the protocol

We will now provide a high level description of RTCMP based on the ideas developed in the first few chapters. The framework provided in Chapter 3 will be used throughout this chapter as the basis for our description. We will fill in the details, to the extent necessary for a high level design, but leaving out system-specific or implementation-specific information. This framework is based on the main tasks common to all fault recovery protocols. They are: detection, instigation, route computation, channel management, and return to normal. All of these tasks need not be supported on all the nodes, though an implementation might be easier to maintain if the same software runs on all the nodes of the network.

5.3.1. Detection

Detection must be supported on all the nodes, since we assume that any link in the network can fail. Thus, each node must monitor its neighboring links for failures. This would be more efficient with some support from the physical layer, for example, posting of events such as loss of transmission clock, physical layer protocol errors, and so on. The software may react to these events, either by handling interrupts posted by the hardware on the occurrence of such events, or by polling hardware status registers at intervals. Events such as retry timeouts of the reliable control message exchange protocol used by RCAP or RTCMP should also trigger detection. The specifics depend on the particular networking hardware and operating system structure, so we will not go into any further detail.

The support from the physical layer may be fast and efficient, but may not catch all faults. Sometimes, a computer may fail without affecting the physical layer's status of its outgoing links. To deal with all failure conditions, as well as to handle networks where the physical layer support may be inadequate, each node must ascertain the status of its neighbors by exchanging "ping" messages, whenever no data has been received on a link for some duration of time. Each node is responsible for ensuring the health of all its incoming links. Thus, if a link fails only in one direction, the node to which data stops arriving would send a ping message and after a few retries would detect the fault. This mechanism is slower than detection supported by the physical layer, and adds some control overhead to the network, but is general enough to be applicable to any computer network. It can detect any fail-stop failures that the physical layer does not detect. It is also general enough to support recovery from node failures, since the fail-stop mode failure of a node can be detected by the neighboring nodes by the ping method as the failures of all the links connecting to the node, even if the physical layer is not affected.

Finally, error rates are another mechanism to detect failures. If a link has very high error rates, so that the quality of the realtime data received from it is unacceptable, it would be better to declare it faulty for realtime purposes, and remove all realtime channels from it. This may be a temporary problem, so that, if the error rate improves after some time, we would return the link to realtime service. Such phenomena may be observed on links established over micro-wave transmission systems, where the error rate changes with the atmospheric conditions. The physical layer may maintain a count of physical layer protocol violations over small intervals, and the RTCMP software should poll this information, and maintain its own filtered counts as described below. If the data delivery protocols implement link level checksums, the errors detected by these checksums should also be counted. This error information may be used as described in (4), normalizing the error rate by the actual data transmission rate in the same interval, and using this metric to declare a failure when the normalized error rate crosses an upper threshold.

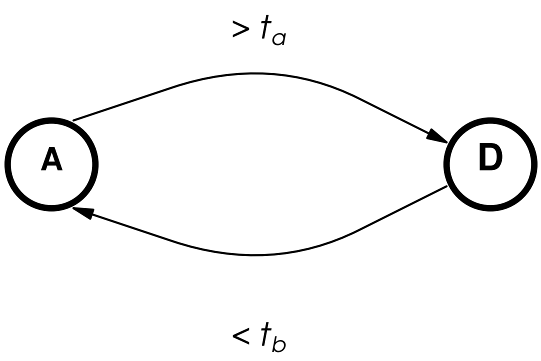

Figure 5.2: Hysteresis loop to avoid oscillations

To reduce the occurrence of oscillations of the link status between "alive" and "dead", with its associated instigation of fault recovery and return to normal actions, some low level validation of the faults detected is essential. A simple hysteresis method is presented in Figure 5.2. A link is classified as faulty if a permanent fault is detected by the other means described above, or if the number of transient errors detected at the physical layer in an interval I_a, expressed as a fraction of the actual data bits transferred, exceeds t_a. However, it is only returned to service if no permanent fault is present and the transient fault ratio measured over a larger interval I_b is less than a threshold t_b. I_a should be small, and chosen depending on the reaction time desired for detection. I_b should be relatively long to avoid oscillations and to ensure that the link is stable before returning it to service. It should be at least as long as the sum of the time to recover from a fault and the time to return to normal after a restoral, since in general we want to avoid declaring another fault on the same link before the return to normal procedure is complete. t_a should be chosen depending on the error rate tolerable for a typical application running on the realtime network. t_b should be at least an order of magnitude lower and may be chosen based on the expected error rates on the transmission medium.

When a fault is detected, the protocol should inform a human network operator to repair the fault, log the event for maintenance and tracing purposes, and also instigate automatic recovery.

5.3.2. Instigation

Instigation of the restoral process is performed by the node that detects the fault. This consists of reliably broadcasting a fault message, containing the link identifier of the failed link, to all nodes in the network so that the routing table in every node reflects the topology after the fault. We need the reliability to ensure that all the nodes receive the same information about the set of failed links. If the underlying network supports reliable broadcast, then this problem is solved. If not, the reliable broadcast is achieved by the RTCMP protocol using a constrained flooding method. The purpose of constrained flooding is to ensure that, as long as the network is not partitioned by a set of faults, the route update messages reach all the nodes in the network. We also want to ensure that messages do not traverse any link more than once. The arrival of the fault message at any node causes it to remove the failed link from the topological database.

For the following discussion of the constrained flooding method, we assume a lower layer transport protocol with checksums and retransmissions to take care of transmission errors. We also assume that clocks are synchronized using a time protocol such as NTP. Each fault message is tagged with a unique fault number, containing the unique identifier of the node and a time stamp. Fault messages have a maximum time to live (*MTTL*), which is a constant known to all the nodes, and which is larger than the maximum possible time to travel across the network's diameter. All RTCMP daemons keep a list of the fault numbers they have seen, and these numbers are not removed from this list until they are older than *MTTL*. Fault messages older than *MTTL* are never forwarded. A message with a fault number which is in the list is also not forwarded. Each RTCMP daemon forwards a fault message, if it is less than *MTTL* old[1] and the fault number is not in the list, to *all* its neighbors. This ensures that the message reaches all reachable nodes, and no link is traversed twice.

The description of the protocol in Chapter 3 also called for a separate reroute message to be sent to the source of each channel affected by the fault. This was needed to support the variety of options we were considering at that stage. However, if we look at the contents of the reroute message in the context of the decisions that have been taken, we find that it only contains the channel identifier of the channel to be rerouted. We can eliminate this message by including the entire set of channels affected by the failure in the fault message. This information must be obtained by querying the local channel management daemon, so the control interface of the channel management software must export a function which takes in a link identifier and returns the channel identifiers of the set of channels using the link. This set of channel identifiers is then included in the fault message broadcast to all the nodes.

5.3.3. Route selection

The route selection function needs to be implemented only on the host machines of the network. Each host node performs route computation for the affected channels originating at that node. To do this, it needs to identify the channels originating locally from the set of channel identifiers carried in the fault message. The RTCMP daemon should query the channel management process to find the set of channels currently originating from the node, and then find its intersection with the set of channels in the fault message. The recovery action described next is performed for this set of channels. Eliminating the reroute message saves any implementation complexity that could have been caused by the arrivals of the fault and reroute messages in different orders.

The process of route selection is performed at the source node, using the Global

[1] if (time stamp > current time - *MTTL*)

locus of reroute constraint, Immediate timing, and Local state prediction. The path selected is the minimum-hop path which has sufficient resources to support the channel and meets the delay constraints. This is computed using the Bellman-Ford algorithm. The information about resources and delay bounds on links is based on the information in the routing database at the time of the computation, subject to the modifications of Local prediction.

In the event of the first attempt failing, the reroute is retried after an interval, which is set to the expected round-trip latency of the network. The route computation during the retry would be based on the route update messages that came in during the delay interval, since the route update protocol would have continued to exchange messages and update the database in the interval. The retry interval should be a tunable parameter of the RTCMP implementation, since the performance of the scheme was seen to be very sensitive to the value of this parameter in our experiments. The retry process should be halted after a fixed interval from the time of the fault, based on the desired maximum response time for the recovery process. This may also be a tunable parameter.

Since the route computation is always performed at the source (the Local routing alternative was rejected due to its poor performance), we can either keep the route computation task in the software for RCAP+DCM or have a separate routing module, as long as the control interface supports the needed functionality.

The general functionality needed in the routing module and in its control interface is described below. Firstly, the control interface of the routing module should allow RTCMP to inform it of the failed links, so that they are removed from consideration during route computation. Secondly, RTCMP needs to specify a set of channels to be rerouted, rather than just one as in the interface provided to the clients. This makes it easier for the routing module to implement the third requirement, which is that the routing software must handle simultaneous route requests using the Local prediction algorithm.

In the specific context of RCAP+DCM, since all the parameters of the channels to be rerouted are already known, and they are not changed during the recovery process, the reroute request interface of the system is simplified. RTCMP just needs to pass to RCAP+DCM the set of channel identifiers. RCAP+DCM can look up the channel parameters from the state stored during the original establishment. This single request subsumes both the routing request and the establishment request, since the RCAP+DCM software performs both actions. We also make one request for all the channels that need to be rerouted at a time, so that the routing software may perform Local prediction on this set of channels. This call *must* be non-blocking, unlike the interface to the clients, since RTCMP might need to react to some other event while RCAP+DCM is performing the establishments. In order to have RCAP+DCM perform Local prediction, we would need to modify the routing software. RCAP+DCM already interacts with the route update protocol to modify the database on which routing in based.

Thus, we can simplify the software structure of RTCMP, as well as minimize the changes that need to be made to the existing realtime network control software, by appropriately extending the interface to the routing module, and implementing the necessary functionality there. The timing and retries are handled in the RTCMP software.

5.3.4. Channel management

The channel management task needs to be implemented on all the nodes of the network on which resources exist to be managed. This includes all the nodes in which RTIP is implemented, as well as any switches which can provide realtime data forwarding without RTIP, but need admission control and resource management support. However, the interface to the RTCMP protocol only needs to exist on the host machines of the network. In the context of RCAP+DCM, since the routing and channel management functions are implemented in the same software entity, the requests for route computation and channel management may be merged into a single call. Channel management, as implemented in DCM, involves a round trip for establishment along the new path, followed by a teardown pass along the old path. In case the attempt fails, RTCMP is asynchronously informed. This causes RTCMP to set a timer event, so that, after the appropriate delay, a retry is attempted. The number of retries is limited, as described in Chapter 4. No channel management action needs to be taken by RTCMP, apart from specifying the reroute requests and retries at appropriate times.

5.3.5. Return to normal

The return to normal task needs to be implemented on the host machines of the network. This action is performed on the detector node, adjacent to the failed link, which also detected the failure. A return to normal is detected when the hysteresis loop of Figure 5.2 restores the link to active status. Since the channels that were rerouted on the detection of the failure may be now on non-optimal paths, we need to reroute them to use shorter paths involving the restored link if necessary. When the link was detected to be in a failed state, the process was time-critical, because the service of the channels on the failed link were disrupted by the failure, and quick recovery was required. In contrast, the return to normal event is not time critical, since no channel is disrupted. If we use the DCM protocol to perform these reroutes, we get the added benefit that, even during the re-establishment process, none of the channels have their service affected. Thus, all that is necessary is to make sure that the rerouting load on RCAP+DCM is not excessive, since this might affect the service seen by other realtime users trying to set up new connections. We want all the channels to eventually move back to their optimal paths, to reduce the realtime load on the network.

On detecting a return to normal, the detector broadcasts the "return to normal" message, which contains the link identifier, to all nodes. The RTCMP daemon on every node contacts the routing module to restore the link into the topology and then triggers a reroute on every locally originating channel, which was rerouted during the fault recovery, or created during the period between the fault and the return to normal. This task is performed at low priority so that it does not affect other channel establishment requests handled by this node. The process looks at all eligible channels and tries to find one for which it can find a shorter route. The source routing algorithm is used without any additional constraint except the end-to-end delay bound. When such a channel is found, the process initiates a reroute onto the shorter path, and then waits for a random time interval with mean T, chosen appropriately to keep the expected channel management load on the network low. The randomization ensures that the retry processes from different nodes are not synchronized. The process is started at the time when the link is restored, and after a time interval of nT, where n is the number of affected channels, all channels would be back on their shortest paths. The process terminates when none of the remaining eligible channels can be rerouted to a shorter path. The DCM protocol ensures that the performance of any of the channels is not affected during the re-establishment process.

This idea can be extended to general load balancing by having this process run continuously at a very low frequency for all channels, rather than only for the channels defined as eligible above.

5.4. Changes to the RCAP+DCM software

The changes which must be made to the control interface of the routing module and the channel establishment software are summarized below.

- Inform routing function of failure (used on all nodes)
- Inform routing function of restoral (used on all nodes)
- Request reroute for set of channels (used on hosts)
- Query to find all channels traversing a link (used on the node detecting the failure)
- Query to find all channels originating from a node (used on hosts)
- Request establishment for channel on a specific route (used on hosts)

The first three features belong to the interface to the routing module, and the last three to the interface to the channel administration software. Since for RCAP+DCM both routing and channel administration are implemented in the same software entity, the third and sixth features may be combined into one call to the combined interface.

The main change needed to the routing software as implemented in RCAP+DCM is the addition of Local prediction. Since the update of the routing database by the route update protocol involves some latency, if the routing protocol is given a number of channels to route simultaneously (as is true for Immediate timing), then the database should be modified after each route is computed to remove the resources that would be reserved by the channel on its forward pass. This is the state that would be seen by the subsequent channels, if they happen to get routed on common links. However, this modified image of the network should be temporary, since the return-pass resource reservations will be lower, and the route update protocol will bring in the correct information, as each channel is established. Thus, in our simulation implementation, we used a temporary copy of the database, which we modified as we computed the set of routes, and finally discarded. The original copy of the database was only modified as the route update messages came in.

The interface to the routing algorithm must also be extended to take into account the fault tolerance schemes; these schemes and the extensions required for them are described in Chapter 6.

5.5. Conclusions

We presented a high level description of a protocol to support rerouting of realtime connections in the event of a fault. The protocol does not discriminate between channels, but attempts to recover as many as it can, within the desired maximum response time of the recovery process. For channels that cannot be rerouted within this time, the recovery protocol informs the user of the failure of the network recovery and tears the channel down. This should only happen when the network is operating close to the maximum realtime load that can be supported.

The recovery schemes designed fit well into the existing realtime network support protocols. The routing software and interface of RCAP+DCM must be modified to some extent, to allow links to be removed or restored by RTCMP, to implement Local prediction for multiple simultaneous route computations, and to support the fault tolerant channels that will be described in the next chapter. The channel management interface of RCAP+DCM must also be modified to support some query functions needed by RTCMP, and to combine the routing and establishment requests for multiple channels into a single call. However, the basic software model remains valid, and RTCMP can perform its tasks as a user of the RCAP+DCM interface.

Many questions have been left unanswered. For example, can multiple priorities for channel rerouting be supported in this paradigm? We can easily extend it to two levels: no recovery support for some channels, and support for other channels, by extending the channel performance specifications to include a flag to indicate this. The above actions would only be performed for channels with the recovery flag on. But to extend this to multiple levels of priority would involve non-trivial extensions.

Another interesting question is related to the interface of the Tenet Suite 2. How can this interface best be extended to provide support for fault recovery? This question too is beyond the scope of this dissertation.

There is also the problem of network security; that is, how do we prevent some unauthorized process from using the RCAP+DCM control interface to reroute channels that it should not be allowed to reroute? This, and other questions, can only answered in the context of a particular implementation, and a specific operating system; hence, we leave the questions open for future work.