Checkpoint Management System

Location

There are two main design decisions for this issue: one would be to have a single checkpoint manager running on every ECU, which is responsible for the checkpoints generated by the RTCR on the respective system. This manager then sends the checkpoints somewhere to be stored. This is nonsensical because for the management system to provide resilience, a failsafe for ECU/RTCR breakdown is needed. Furthermore this kind of functionality could have been implemented directly into the RTCR in the first place. Therefore we propose a centralized management system which receives checkpoints from the RTCR components on their respective ECUs and both stores them and is responsible for their migration and restoration on another system. This however marks a single point of failure in the entire infrastructure, reducing system resilience and making the manager pointless. Thus redundancy is used in the form of at least two checkpoint management systems running on different ECUs.

Receiving of Checkpoints

Publish/Subscribe

The first thing that comes to mind to resolve the issue of sending the checkpoints on the side of the RTCR and receiving them on the side of the CMS is a simple publish and subscribe system, where all instances of RTCR connect to a socket set up by the manager and then send serialized checkpoints over ethernet whenever a new one is created. This approach is commendable for its simplicity of both concept and implementation, but has a couple of flaws: first of all, because of the redundancy of management systems the manager is not transparent to the RTCR because checkpoints have to be sent to every instance of the checkpoint manager. Another major problem is that for many RTCR components which in turn manage many components, lots of network traffic in form of checkpoints is sent to only one interface of the centralized checkpoints management system. This will quickly lead to network congestion. Therefore a solution where the CMS can receive checkpoints at its own pace is desired and presented in the following section.

DSM Weidinger

Weidinger implemented a distributed shared memory for Genode 15.02 which spawns broker nodes from the components which aim to share memory, which then connect over TCP/IP. The codebase unfortunately is written for an old Genode API and therefore require extensive updating. In general, the approach of a DSM works well for our use case since it makes it possible to exchange checkpoints between one RTCR instance and the at least two redundant instances of CMS transparently, since the RTCR theoretically never directly has to address the CMS. For the case of CMS and an instance of RTCR operating on the same ECU, a simple shared memory using Genode dataspace capabilities is used. The CMS checks for this case by handling an error that the broker should throw when it is asked to connect to a remote broker that is located on the same machine. A more elegant solution would have the CMS compare the IP addresses of the local and remote brokers, but an interface to query the IP addresses is not provided. This would be solved by integrating the DSM functionality directly into RTCR and CMS. Once the DSM is established CMS instances would then have to monitor the dataspace and store a checkpoint to their own memory whenever a change is detected. Unfortunately this kind of detection is unfeasable as the CMS itself never has direct access to the memory the RTCR writes to. Two possibilities of resolving this issue are discussed. Firstly, there is the simple solution of serializing and downloading all new checkpoints at set intervals. The advantage of this approach is its simplicity, but it suffers heavily from the chance of increasing checkpoint age when having to fallback farther than the granularity of the RTCR checkpointing interval. Mitigating this issue by increasing the interval entails massive network load, as a whole array of checkpoints is downloaded frequently. Another, completely different solution for the receival of checkpoints, which doesn't rely on downloading intervals could also prove beneficial: here the RTCR broadcasts a notification whenever it writes a new checkpoint to its shared memory, containing the memory address of the updated checkpoint, resulting in a hybrid between publish/subscribe and DSM. The CMS can then use the IP address from the IP-header, finds, serializes and downloads the checkpoint in a newly spawned thread. One advantage is that the RTCR doesn't have to interrupt its proceedings to send a checkpoint, as it simply writes to memory and the transmission of the snapshot is then handled by the broker, inproving transparency. Another argument for this approach is that the congestion noted in the section describing a possible publish/subscribe solution is distributed to all the brokers spawned by the manager. Therefore it might seem like the obvious choice, but there exists one caveat: to maintain transparency of the redundant managers, every RTCR has to broadcast a small amount of data whenever a checkpoint is made, containing possibly sensitive information. For many RTCRs handling many components the amount of messages might overstress the network. This is only an issue in the current implementation, as it is resolved when the DSM functionality is integrated into is participants: because of the integration, the RTCR knows of the IP addresses of its participants, and can thus replace the broadcast with a multicast. Transparency of course suffers a little because of this hybrid solution, but all in all is a negligible problem as the addresses of the checkpoint management systems are only used for this single multicast, which can be implemented in a completely flexible fashion concerning the number of redudant checkpoint managers. Furthermore addressing of the management systems is necessary anyway in scope of migration and restoration, as outlined in the eponymous section. The reason which ultimately led to the decision to implement this approach over the simple downloading solution stems from another problem which might occur in the future: the brokers on the side of the CMS might allocate too much memory, as one broker is spawned for each DSM with another machine. This introduces a scalability issue into our system, which could be mitigated using a solution, which relies on the aforementioned hybrid of publish/subscribe and DSM for checkpoint receival. Instead of RTCR using the DSM for the entirety of its checkpoint storage, it stores them locally and uses the DSM only for passing of checkpoints to the CMS. Here it writes the checkpoint to the DSM, sends a notification and then waits for both management systems to download the checkpoint. Knowledge of the completion of the download is passed to the RTCR via network, after which the RTCR frees the DSM on its side. Currently this wouldn't have the desired effect as freeing up dataspace on one side of the DSM doesn't do so on the other side, because an comprehensive consistency module is not part of Weidinger's DSM. Furthermore the DSM would need to return the now unnecessary memory to its parent. Both of these extension should be part of an integration of DSM directly into RTCR and CMS, which is further discussed in future work. For the scope of this thesis however, especially the consistency is a non-issue, since only one participant of the DSM writes into the shared memory and enough memory iis available.

Storage of Checkpoints

The checkpoints will be stored by both management systems on the same network addressed storage in RAID configuration. We choose to have both the main and the redundant system write to the same memory, because at it is already planned to be in RAID configuration, the possibility of single point of failure is mitigated and therefore it is not necessary to integrate two network addressed storages into our infrastructure. For purpose of efficient retrieval the CMS has to know both the ID of the program of which the checkpoint is stored and what ECU it came from, preferably using MAC address. This ID originates from the RTCR which, since it protects multiple programs, has to distribute IDs at some point. The MAC address of the ECU can either be stored by the RTCR in some global field or be queried everytime a checkpoint is sent to the CMS. The ID, MAC and serialized checkpoint will then be composited into a class containing all the information.

Migration & Restoration

There are two different scenarios where a migration and restoration is necessary: the first one is when one of the RTCR components asks the CMS to migrate and restore one of its programs, for example for load balancing reasons. In this case the MAC address and ID of the checkpoint have to be transmitted, because duplicate IDs are possible, since they only have to be unique on a single ECU. The CMS then retrieves the serialized checkpoint from memory and selects the ECU to restore on.

The second case is that the ECU or the RTCR thereon stops working entirely. For this purpose we assume that the CMS knows of the event and the MAC address of the board it occured on. We then retrieve all checkpoint objects with that MAC address and restore them one by one on another selected machine. The selection for both scenarios happens by comparing the ECUs in the network using different performance metrics. These metrics include:

- CPU load
- RAM usage

of the ECU. Since this information is not accessible through any already existing Genode interface over network, the RTCR has to provide such an interface, where upon demand it accesses CPU load and/or RAM usage locally.

One big issue is that the redundancy of managers warrants that they have to agree on who performs the restoration to avoid duplicate processes. Here we have to acknowledge the possibility of the managers migrating the process to different platforms, as timing differences of the managers might make one ECU more suitable than it was at migration time of another checkpoint manager. The decision is therefore made at the time of checkpoint retrieval. Whoever is the first to access a checkpoint from the NAS is the one to migrate and restore the checkpoint, which is enforced by a mutex and the deletion of the checkpoint from the database. This was necessary anyway, because both ID and MAC-address are those of the old host of the process and therefore expired.

Not Scope of the Implementation

Knowing a Restoration is necessary

Although it is not in the scope of the practical work of this thesis, some discussion on the topic of actually knowing when a restoration is necessary should be done. Most cases of migration and restoration happen for reasons where the RTCR decides that a migration is sensible. We then receive an opcode, MAC address and ID of the checkpoint, migrate and restore. The actual failsafe and resilience improving component of our infrastructure, which was the main motivator to actually implement such a checkpoint manager, however is more complicated. The first, quite naive approach is to regurarly ping every board in the system and assume ECU failure when a certain number of pings don't return. This would not only create unnecessary traffic on the network but also mean that the CMS would need a thread running for every board from where these pings are made. Overall the network traffic and CPU load are probably not feasible for an actual real world distributed system. The only other way to fix this is to have some overarching entity constantly monitoring all ECUs and sending a signal to the CMS whenever a breakdown occurs. This then still leaves the problem of the RTCR failing with the ECU still being intact.

Managing Checkpoints in Real Time

An important aspect of RTCR is its real-time capability. Therefore a discussion concercing the real-time capability of the system managing this RTCR has to be made, even though it is not part of the practical implementation. There are two aspects that may hinder the target child at fulfilling its schedules. One of them is the obvious lost time of having to migrate and restore the program, which has to be combated by increasing performance of migration and restoration, maybe even using hardware-assisted solutions. One could be for example to identify when a program to be restored is on a particularly tight schedule and thus restore it on the machine the CMS itself is running on. The second aspect is the schedule on the machine that is being migrated to. If many programs with real-time demand are already running, the scheduler on the new ECU might have to enforce that our program misses a schedule. Mitigating this completely would entail knowing the state of the scheduler on any ECU that is eligible for migrating to, but an estimate can already be made using CPU load and RAM usage, which is already included in the ECU selection. All in all a new interface would be required and the information that is retrieved from this interface would then furthermore influence the selection of suitable machines to migrate to.