ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV033

DISTRIBUTED TRANSACTIONS IN CERN'S ACCELERATOR CONTROL SYSTEM

F. Hoguin, S. Deghaye, P. Mantion, J. Lauener, R. Gorbonosov, CERN, Geneva, Switzerland

Abstract

Devices in CERN's accelerator complex are controlled through individual requests, which change settings atomically on single Devices. Individual Devices are therefore controlled transactionally. Operators often need to apply a set of changes which affect multiple devices. This is achieved by sending requests in parallel, in a minimum amount of time. However, if a request fails, the Control system ends up in an undefined state, and recovering is a time-consuming task. Furthermore, the lack of synchronisation in the application of new settings may lead to the degradation of the beam characteristics, because of settings being partially applied. To address these issues, a protocol was developed to support distributed transactions and commit synchronisation in the CERN Control system, which was then implemented in CERN's real-time frameworks. We describe what this protocol intends to solve and its limitations. We also delve into the real-time framework implementation and how developers can benefit from the 2-phase commit to leverage hardware features such as double buffering, and from the commit synchronisation allowing settings to be changed safely while the accelerator is operational.

DISTRIBUTED TRANSACTIONS: INTRODUCTION

In CERN's accelerator control system, clients address devices individually with a request made on a property of a device. To modify a property across multiple devices, requires as many requests as there are devices. Until all the requests are made, an accelerator is in an undefined state. Moreover, if any of the requests fail, additional actions are required to understand why it failed and how to fix it, leaving the accelerator in an undefined state for a potentially large amount of time.

In order to solve this problem, distributed transactions were designed and implemented in CERN's accelerator control system.

GOALS, LIMITATIONS, AND DESIGN OF DISTRIBUTED TRANSACTIONS

The goal of distributed transactions is to ensure that a set of modifications occurring on distributed nodes are either completely recorded, or not at all. A node can be anything, but the first example that comes to mind is a database. This behaviour is typically achieved by using a two-phase commit. The generic workflow is as follows:

- A transaction is opened with the different participating nodes;
- Modifications are made on the different nodes in any order and over an unspecified period of time;
- The commit then occurs in two steps:
  1. A commit is sent to all the nodes which perform the required checks and ensure that locally there are no errors;
  2. If no errors are reported by the first commit, a second one, confirming the wish to commit is sent. From then onwards, the modifications are permanently recorded;
  3. If, on the other hand, errors are reported during the first-phase commit, the transaction is rolled back on all the nodes and no modifications are recorded.

DISTRIBUTED TRANSACTIONS IN ACCELERATOR CONTROLS

In the context of CERN's accelerator controls, the nodes are the low-level equipment controllers, the so-called Front-End Computers (FECs). The modifications are changes of the underlying equipment settings. A nominal transaction for CERN accelerator controls contains the following steps:

- The client opens a transaction with a unique transaction ID on all the devices it plans to modify, via a synchronous middleware call to a standardised property;
- The client modifies the settings via a synchronous middleware call. Again, the order and period of time are unspecified;
- The client asks the devices to test their new settings via a synchronous middleware call to a standardised property (TRASACTION.TEST);
- If all the devices report a successful test, the client commands the timing system to broadcast a commit event for the transaction. Upon reception, the FECs permanently commit the new settings;
- Otherwise, if one or more devices report an error during the test, the client asks the timing system to broadcast a roll-back event which will trigger the discard of the settings set with the corresponding transaction ID.

In the workflow described above, several implementation choices are already visible. One key element is the way to transport the final commit and roll-back. CERN's timing system, whose purpose is, among other things, to distribute events to the whole accelerator complex, is used to trigger a commit or rollback of a transaction. Even though the transactions are not meant to synchronise the moments at which the commits are performed, by using the timing system to transport these two events, we combine the possibilities offered by a global commit event with the synchronisation possibilities of the timing system.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV033

In addition, mainly for test purposes, the client may trigger a commit or rollback event without using the timing system. This is done by telling every device in the transaction to commit or rollback via another middleware call to the TRANSACTION.COMMIT or TRANSACTION.ROLLBACK properties. It is acknowledged that in this case, the commit loses its atomicity, since the client is limited to regular TCP connections to do this, without any multicast possibility.

It should be highlighted that, on the low-level equipment controllers (FECs), a single server typically handles many devices. Clients do not use this information in any way and must treat each device regardless of the FEC they are handled by. However, this has some implications on the design of the server-side commit process. The implementation of the transaction protocol in a device server, where a transaction is processed as a whole entity (in opposition to device by device), is explained in the following sections. Why such a design choice was made, its constraints, and how the technical challenges were solved are also described.

DISTRIBUTED TRANSACTIONS IMPLEMENTATION IN FESA

The Front-End Software Architecture (FESA) framework is used to do real-time programming to control accelerator hardware in CERN's control system. It offers the possibility to react to specific events through an API giving access to all the concerned devices at once. Many systems developed with FESA are a composition of multiple devices, and if settings are changed on multiple devices within a transaction, this must be seen as one change of all the settings, rather than multiple changes of individual devices.

Since clients are not aware of this (they always treat devices as individual entities), this must be transparently implemented in FESA. In addition, a FESA device server can be run either as a single process (see Fig. 1) or as two separate processes, called server process and real-time process, which interact through a shared memory and several message queues (see Fig. 2).

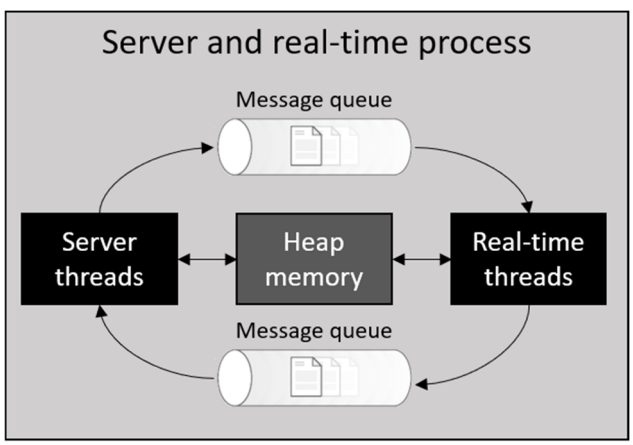

Figure 1: FESA server running as a single process.

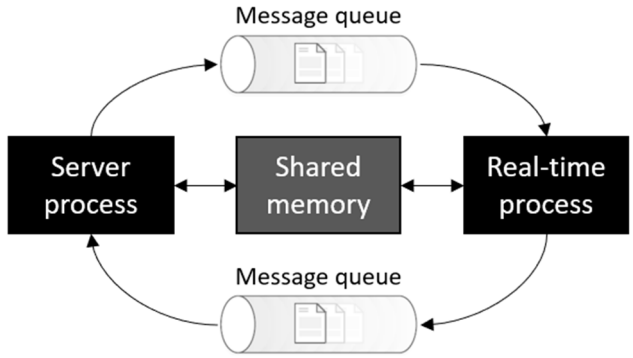

Figure 2: FESA server running as two processes.

This has several implications:

- FESA must be able to uniquely identify and pass the data from a transaction across processes.
- Since the transaction is opened sequentially on all the devices by the client, FESA cannot know in advance the size of the settings that will be modified in the transaction.
- When a transaction is tested by the client, test requests will happen sequentially on every device. FESA must test the full transaction on the first received test request and ignore subsequent test requests on other devices.

Many failures can happen during a distributed transaction (hardware error, device server crash, network issue, timing system issue). A commit event is thus not guaranteed to be processed by all the devices in the transaction. Because of this, the processing of the commit event by the devices should be as small an operation as possible, e.g., a buffer swap. Devices must use the test request to prepare for an upcoming commit or rollback. Consequently, the implementation does not guarantee the state of the system in case of a failure during the commit. Furthermore, due to the nature of the timing system, it is difficult to provide feedback to the client in such a case. Some solutions exist to mitigate this problem [1], at the cost of an increased complexity.

Finally, in order to provide a transaction implementation across as many devices as possible, it was decided that FESA would offer two modes:

- A transparent mode, with no code modification required by the users, but without the possibility to leverage some hardware capabilities such as double buffering. This is intended for cases where validation of settings is done entirely in software, which guarantees that commit events will successfully apply the new settings unless an unpredictable hardware error occurs.
- An advanced mode, which requires the user to implement entry points for test, commit, and rollback. In this mode, the test event can be used to prepare the hardware, for example by preparing a

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV033

buffer containing the new settings to apply, and the commit event can be used to trigger a buffer swap, applying the new settings atomically in a very short time.

These two modes are mutually exclusive in a FESA process.

TRANSACTIONS IMPLEMENTATION IN FESA: TRANSPARENT MODE

In order to fully understand the implementation of transparent transactions, it is first necessary to understand how a normal modification of a setting works (outside of a transaction):

- The client sends a new setting.
- Its validity is checked by some user code, which may accept or reject the new value.
- If the new setting is valid, the memory containing the setting values is updated with the new value [2]. If the setting needs to be applied immediately, an event is sent to the real-time FESA process, possibly across process boundaries if running in split mode.
- The event is received and processed by the real-time part, which applies the new setting on the hardware.

If the setting is instead modified within a transaction, the sequence is as follows:

- The client sends a new setting.
- Its validity is checked by some user code.
- If the new setting is valid, its value is saved in a buffer dedicated to this transaction. If the setting needs to be applied as soon as the transaction is committed, an event is prepared and saved in the transaction buffer.
- The client sends other settings, and the same process is repeated until a commit or rollback event happens (in transparent mode, the test event is ignored, as the validity of settings has been checked already).
- If a commit event happens, the memory containing the setting values is updated with the buffered setting values. All the buffered events are triggered.
- The events are received and processed by the real-time part, which applies the new settings on the hardware.

Thus, in transparent mode, the modification of settings within a transaction are equivalent to modifying all the settings in rapid sequence.

While the transparent mode offers the main advantage of being usable without user code modification, it does not cover one key element of the nominal sequence which is the possibility for Sets to be done in an unspecified order. To support this, a more sophisticated implementation is required, the advanced mode.

TRANSACTION IMPLEMENTATION IN FESA: ADVANCED MODE

The goal of advanced mode is to allow the user to benefit from the 2-phase commit (test, then commit) to test new settings only after they have all been set, and to leverage the capacity of the controlled hardware to prepare for a commit.

A fictitious but simple example (which can be applied to many other use cases) can demonstrate why this is important:

Consider a lab-bench power supply, which has a power output limit of 300 watts. It is connected to a FESA process which can modify its voltage and maximum current through a serial connection.

When a new voltage or maximum current is requested by a client, the user code must check that the product of the two does not exceed 300 watts. Assume that it is currently configured to 5V and a maximum of 20A.

If a client wants to set it to 30V and 5A, without the transactions, it must be careful to first set the maximum current, otherwise the maximum power will exceed 300W in the transition (30V times 20A). This requires the client to know about this logic, which is not feasible in the case of automated settings management.

The transaction implementation in advanced mode solves this problem by delaying the validation of the settings until the test request happens. The user code can still reject a setting value as soon as it is sent if it is invalid on its own (e.g., setting a voltage of 10000V), and in addition, it can check the validity of the settings combination in the code that is called when a test event happens. The API provided by FESA in that case transparently provides the values stored in the transaction buffer if present, or previously known otherwise.

Now also assume that the power supply can receive its new voltage and maximum current settings but wait for a further command to apply them. In addition to ensuring the settings validity, the implementation of the test event would then send the new values to the power supply. If anything goes wrong at this point, it is still possible to signal a test failure to the client, who can roll back the transaction, or try with other setting values.

Finally, when the commit event arrives, the only thing left to do is to tell the power supply to apply the new settings. The possible failure surface has thereby been reduced to the strict minimum, allowing the clients to cancel the transaction if anything else goes wrong.

While this example is voluntarily simple, it applies to a lot of hardware used in accelerators, such as power converters, digital oscilloscopes, and much more.

ADVANCED MODE OUTSIDE OF A TRANSACTION

As explained above, transactions in advanced mode require the user to adapt their code to handle the test, commit and rollback events. However, the settings can still be changed outside of a transaction, in which case they need to be applied immediately.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV033

For such cases, FESA simulates a transaction: the request to modify a setting will cause FESA to generate a test event, and if it is successful, a commit event. If the test event fails, the new setting value will not be applied, and the client will receive the test event error message, informing why the new setting could not be applied.

While this approach may seem complex at first, it has many advantages: the complexity remains in the FESA framework, where it is thoroughly tested. The user only has to write a single implementation of the code which communicates with the hardware, and as a result only has to test one code path.

TRANSACTIONS ACROSS PROCESS BOUNDARIES

The FESA framework may be used in what is called "split mode". This is meant for highly critical systems, which should always be up and running. In this mode, the real-time part of a FESA deployment is launched in a process, while the server part, responsible for handling client requests, runs in a separate process. This can be seen as a way to sandbox the real-time process.

The two processes communicate using standard IPC methods: shared memory, message queues, shared mutexes and condition variables. From a user point of view, nothing changes feature-wise, but there are a few restrictions, mainly due to the fact that pointers are not valid across process boundaries.

In order to implement the transactions feature in FESA, these considerations had to be taken into account. As a result, the transaction buffer, allocated when a transaction is opened, is a dynamic structure which cannot contain pointers. The devices and their settings are given unique indices, allowing to identify them across the two processes of a split mode deployment. Every transaction buffer also contains a mutex and a condition variable, which are used to signal the completion of the test, commit, and rollback events.

There is another consequence of running in split mode: the memory allocated for the transaction buffer must be contiguous, as it is mapped between the two FESA processes using the *mmap* system call, which takes an address and a size. However, it is not possible to know in advance how much memory will be needed for the transaction buffer: the client opens the transaction sequentially on an unpredictable number of devices. Furthermore, it is allowed to add devices to the transaction as long as the test request has not been sent.

To solve this problem, a naive solution would be to allocate all the memory potentially needed immediately, by computing the maximum size of the transaction, which corresponds to the size of all the settings of all the devices in the server, plus the size required for the transaction's bookkeeping structure. However, many FESA processes operate on machines which have a low amount of memory, without enough memory left to do this. This means that it would be impossible to conduct a transaction on such a server, even if it would affect a single device.

The solution is to take advantage of the MMU of the CPU, and the capabilities of the operating system: it is possible to reserve a chunk of contiguous memory in a process, without actually mapping it to any physical memory. This works even with memory shared between processes, by using the appropriate flags in the call to *mmap*: with PROT_NONE as protection together with the MAP_PRIVATE and MAP_ANONYMOUS flags, the memory is marked as inaccessible, causing the operating system to not allocate any physical memory, but to still reserve the address space in the process, guaranteeing a contiguous memory block. When the buffer needs to be extended, and some actual physical memory allocated, *mmap* is called again, allowing to allocate an arbitrary amount of the reserved memory. This time, we pass PROT_READ and PROT_WRITE to make the memory readable and writable, together with the flags MAP_FIXED and MAP_SHARED, ensuring respectively that the address of the reserved memory is not changed, and the memory is visible in other processes.

Tests on multiple Linux platforms have shown that this works as expected. On Windows platforms, the same effect can be obtained by calling *VirtualAlloc* with the flag MEM_RESERVE, and then MEM_COMMIT.

CONCLUSION

The design and implementation of distributed transactions in CERN's accelerator control system have been described. During the design process and for simplicity reasons, a choice was made to not make it resilient to hardware errors, but instead to allow the detection of errors until as late as possible before the commit happens. The reason is that, unlike a database, if a hardware error occurs, rolling back to the previous state is not necessarily possible, nor desirable.

Because of this, implementing a strong guarantee on the atomicity of transaction commits is not possible. Any attempt at doing so would be much more complex than the presented implementation and would require compromising on other aspects, such as the time needed to process a commit. In addition, keeping the distributed transaction design relatively simple ensures that it is possible to implement on any device. In practice, this allows to offer in FESA both a "transparent mode", allowing users to enable transactions on their devices without having to write a single line of code, and an "advanced mode", where users can fully benefit from the flexibility of the 2-phase commit.

Nevertheless, given the nature of the commit and rollback events, a logical next step is to implement a system of feedback to allow clients to know immediately if a commit failed, and why. Since the commit and rollback events are broadcast by the timing system, this requires an extension of the protocol to allow clients to be notified when that happens.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-TUPV033

REFERENCES

[1] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," in *Proc. VLDB'14*, Hangzhou, China, September 2014, pp. 181-192.

[2] F. Hoguin and S. Deghaye, "Solving the Synchronization Problem in Multi-Core Embedded Real-Time Systems", in *Proc. ICALEPCS'15*, Melbourne, Australia, Oct. 2015, pp. 942–946, doi:10.18429/JACoW-ICALEPCS2015-WEPGF102