

CERN Summer Student Project Report

Molr - A delegation framework for accelerator commissioning

Submitted by
Nachiappan Valliappan

Supervised by
Marc-Antoine Galilee
Jean-Christophe Garnier



CERN, TE-MPE-MS

June - August 2017

Abstract

Accelerator commissioning is the process of preparing an accelerator for beam operations. A typical commissioning period at CERN involves running thousands of tests on many complex systems and machinery to ensure smooth beam operations and correct functioning of the machine protection systems. AccTesting is a software framework which helps orchestrate the commissioning of CERN's accelerators and its equipment systems. This involves running and managing tests provided by various commissioning tools and analyzing their outcomes. Currently, AccTesting only supports a specific set of commissioning tools. In this project, we aim to widen the spectrum of commissioning tools supported by AccTesting by developing a generic and programmable integration framework called Molr, which would enable the integration of more commissioning tools with AccTesting. In this report, we summarize the work done during the summer student project and lay out a brief overview of the current status and next steps for Molr.

Contents

1	Overview	1
2	Technical Objectives	3
2.1	Simple & Generic API	3
2.2	Delegation	3
2.3	Control & I/O	4
2.4	Security & Focus of development	5
3	Work Done	6
3.1	Communication layer	6
3.1.1	Implementation & Terminology	6
3.1.2	Molr use by Developer	8
3.1.3	Molr use by Client	9
3.1.4	Client API & Molr Server	10
3.1.5	Internals: Remote execution	10
3.1.6	Internals: Input/Output	11
3.1.7	Internals: Execution Control	11
3.1.8	Internals: Error propagation	11
3.2	Infrastructure layer	12
4	Conclusion and Future	14
	Acknowledgements	16
	References	17

Chapter 1

Overview

The Large Hadron Collider (LHC) is a large and complex machine which involves many sub-systems which are required to work in cooperation during beam operations. These systems are qualified periodically during dedicated commissioning periods and retested after corrective or regular maintenance. Running tests on a large number of systems is a complex task especially when considering the constraints which need to be kept in mind while testing a specific component. For example, system C might need to be tested only after qualifying systems A and B. AccTesting accepts user-defined tests (or commissioning sequences) provided by various commissioning tools and executes these tests considering constraints which have been specified by the user.

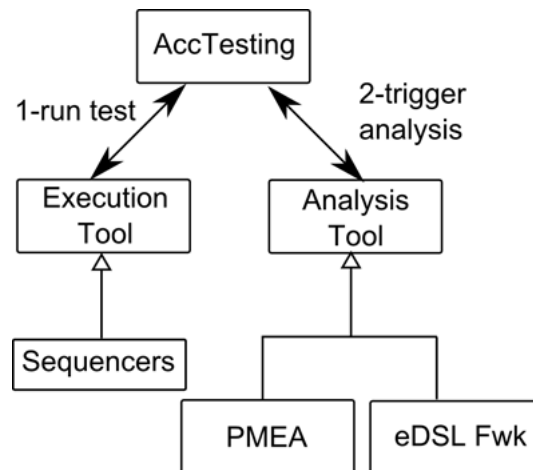


Figure 1.1: AccTesting framework

AccTesting serves both as an extensible software framework for facilitat-

ing automated commissioning sequences, and as a unified interface to orchestrate, manage and monitor tests [1]. Although the execution of a test can be triggered using the user interface, AccTesting itself does not execute these tests. It delegates the execution of these tests to an "Execution Tool" shown in Figure 1.1. This follows from the aim of being generic enough to enable integration of new execution and analysis mechanisms. Figure 1.1. provides an overview of the AccTesting tool suite. The "Execution Tool" branch represents the umbrella of all supported execution tools and the "Analysis tool" branch represents the umbrella of all supported analysis tools. Currently, the Sequencers are the only execution tool which provide a specific set of commissioning tests. In this project, we aim to add a new generic execution tool which would in turn be able to integrate any new commissioning tool! Figure 1.2 shows a Molr included ecosystem of AccTesting.

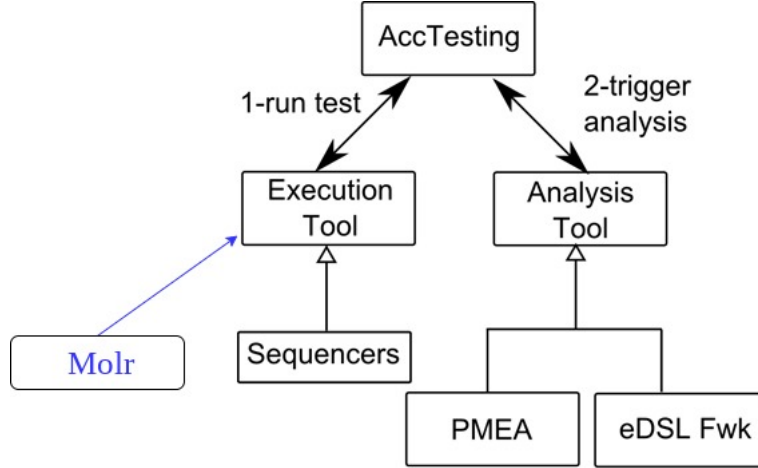


Figure 1.2: AccTesting framework, with Molr

In effect, Molr aims to be a a programmable software framework for integrating new commissioning tools and to provide an interface for delegated execution of tests supplied by these commissioning tools.

Chapter 2

Technical Objectives

The primary goal of Molr is integration of new commissioning tools into the AccTesting ecosystem and delegated execution of tests provided by these tools. This chapter lays out the technical objectives of Molr which help achieving the larger goal. These objectives were formulated, analyzed and discussed as a part of this project.

2.1 Simple & Generic API

Commissioning tools are targeted at many different systems and developed by the systems experts, hence they can widely vary in nature. The Molr API must be generic enough to avoid limiting the integration to only a certain *kind* of commissioning tools. While aiming to be generic, we also need to keep the API simple enough to enable easy integration of commissioning tools. It would be cumbersome to perform complex and time consuming set of changes to a commissioning tool in order to be used with Molr, hence simplicity of the API plays a key role in being able to reuse the code of existing commissioning tools. In short: The Molr API needs to be simple, but without the loss of generality.

2.2 Delegation

Depending on the resources required by a set of missions run during a commissioning campaign, it may have to be executed and managed on a dedicated cluster of nodes. In other cases, missions may have specific requirements: it might be possible to execute some missions only inside a specific network or on a specific server.

This target location - of where exactly the mission is executed - should not be a concern of the client (AccTesting). The Molr server must provide complete location transparency of test execution.

To make delegation possible, especially since the tests might be of varying nature, we expect them to be individually executable entities which completely package all their required dependencies. For example, the test to be performed on a specific hardware component is expected to package the implementation of the communication with the component's software interface. This assumption allows Molr to simply treat tests as executable entities and execute them anywhere in an authorized environment.

Consider the case of closing the frequency test loop in the Beam Interlock System (BIS): it involves interacting with dozens of devices of different kinds (CIBMs, CIBGs). This test is expected to bundle all that is necessary to communicate with these BIS devices, without the Molr Server or supervisor providing domain specific tools/libraries or information.

Another way to formulate this objective is to call Molr a *delegated execution manager* for test execution with complete location transparency. Viewing Molr as a service to run tests brings up an important question of security and safety, this is discussed in later sections of this chapter.

2.3 Control & I/O

The commissioning tools (which contain tests) are written by a developer, while the commissioning process is carried out by an operator from the CERN Control Centre (CCC). In addition to being able to execute tests, the operator should have complete clarity and control over the test execution.

Control and clarity are crucial in commissioning certain systems. For example, the LHC collimator commissioning sequence is a series of steps: checks (collimator corner orientation, jaw movement, position sensors), analysis (data from temperature sensors) and calibration (jaw width, LVDT calibration). If the commissioning fails at a certain step, an error trace must be reported to the operator. In addition to an error trace which indicates the possible cause of failure, the operator should be able to *step through* the test and identify the specific step causing the failure of the test.

Stepping is more than just debugging. The ability to step through a test plays a crucial role in understanding and analyzing a test execution. Each line of code in a test may potentially communicate with many systems, in which case, the operators may want to closely look at the side effects caused in other system supervision applications, or may want to interrupt a test procedure for an expert of the system to analyze the cause. Stepping enables

fine grain control, and could be used the first time the test is run or after any changes are performed in the system under test.

In addition to control, Molr must also facilitate the communication between the operator and the execution of a test, i.e, collect input from the operator and provide it to start test execution and collect results from a test and send it back to the operator. In case of failure, an error must be sent to the operator with sufficient details. While most programming languages offer these features out of the box, ensuring that these objectives are achieved while providing complete location transparency is a non-trivial task. This involves transmission of serialized application state (for input or output) and/or error traces/exceptions (in case of test failures) from the machine which actually runs the test to the client and vice-versa.

2.4 Security & Focus of development

Given the above objectives, an important question arises: How does one ensure that an unauthorized person does not submit and execute a malicious commissioning sequence?

For the most part, this remains an open question and is yet to be fully defined. The answer to the question also depends largely on how the test deployment (or submission) interface to the Molr service is defined. One quick way to reduce potential risks of safety, is to ensure that the Molr service is accessible only on the technical network (TN) from a specific set of white-listed sources. This can be easily done at an administrative level outside the framework's implementation. Additionally, since the current version does not provide an automatic way to deploy a mission, these risks are automatically mitigated by supporting only a strictly defined set of missions which require manual intervention to deploy.

This project concerns itself with the design and development of Molr focusing on laying the foundations for location transparency and the internal communication. This effort is also aimed at showing that it is possible to execute *any* commissioning sequence in a distributed environment (a consequence of remote test executions) while facilitating input/output and complete control of test execution.

Chapter 3

Work Done

Figure 3.1 provides a high level overview of the Molr Architecture. The blue arrows represent the *Infrastructure layer* and the red arrows represent the *Communication layer*. The implementation in this project focuses primarily on the fully specified Communication layer. The Infrastructure layer requires further thought and a closer analysis of requirements when considering the security implications and actual development workflows of commissioning tools. Some investigations made as a part of establishing the foundations for the infrastructure layer are also discussed later in this report.

3.1 Communication layer

The Communication layer is concerned with the communication between the three main components: the *Client API* (the programmable component), the *Molr Server* (the service component) and the *Mole supervisor* (the test execution component).

3.1.1 Implementation & Terminology

Molr is a Java framework. The Client API is offered in Java since most of the software which would use it (commissioning tools & AccTesting) is written in Java. Considering inter-operation between the API implementation and components, and taking into account the prevalent use across CERN, Java was chosen to implement all the components. The internal communication between Client API and the Molr Server takes place using a REST-like Web API offered by the Molr Server. Molr Server is a Spring Boot application and relies on Spring's serialization and de-serialization capabilities to implement the Web API.

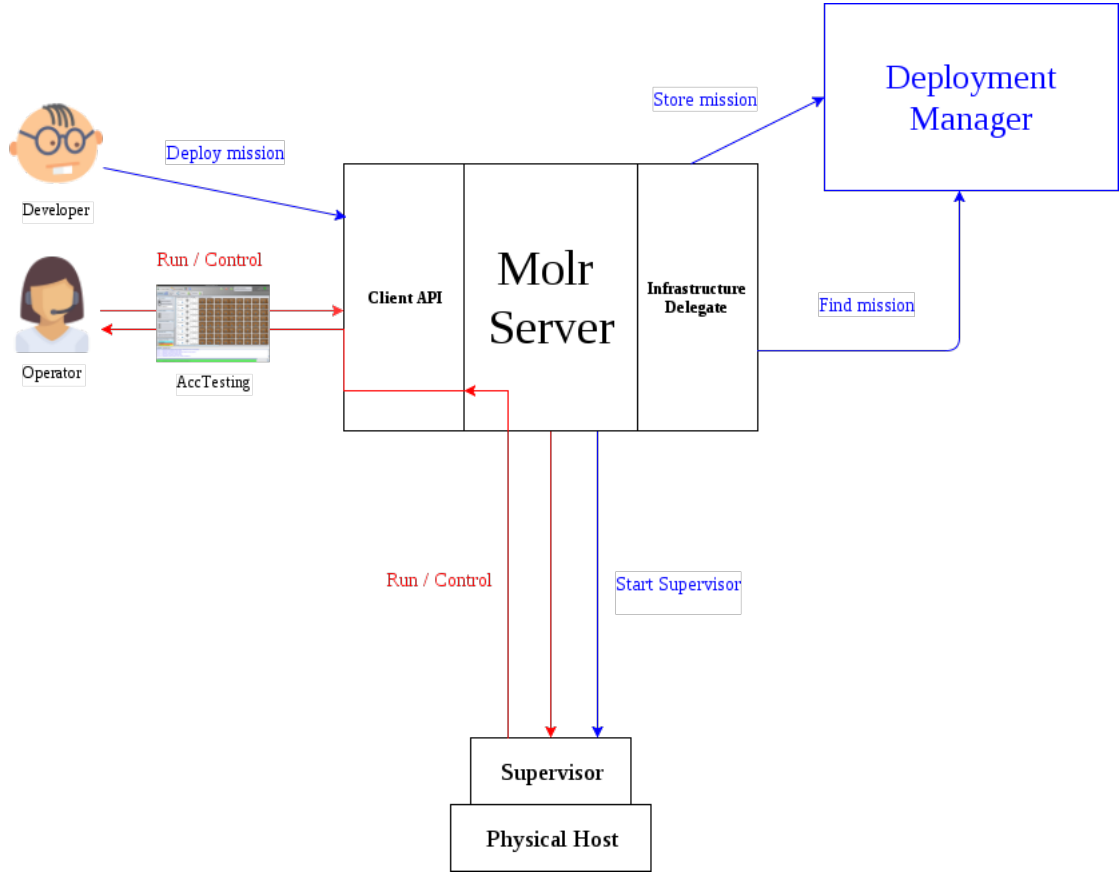


Figure 3.1: Molr Architecture

Molr uses different terminology from the one used in the context of accelerator commissioning. The reason is that Molr is in no way strictly tied to accelerator commissioning; it solves a generic problem of running and controlling executable code in a distributed environment. To enable re-usability of the solution in other contexts, the terminology used is different.

A *Mission* is a class which contains executable code (in our case tests). A *Mole* is an interface which can be implemented with the logic needed for executing a Mission. For example, a `RunnableMole` class (which implements the `Mole` interface) knows how to run a Mission class which is an implementation of the `Runnable` interface. A developer must bind a Mission with an appropriate Mole. This is done by simply adding an annotation in the Mission class (Shown in Figure 3.4).

The concept of a Mole offers the developer with the flexibility to define *how* a Mission should be executed. For example, `IntegerFunctionMole` in Figure 3.3 knows how to execute a class implementing the `Function<Integer,Integer>`

interface (implementation not shown here).

```
public interface Mole<I, O> {  
    List<Method> discover(Class<?> classType) throws IncompatibleMissionException;  
    O run(Mission mission, I args) throws MissionExecutionException;  
}
```

Figure 3.2: The Mole interface

```
public class IntegerFunctionMole implements Mole<Integer,Integer> {  
    public Integer run(Mission mission, Integer arg) throws MissionExecutionException {  
    public List<Method> discover(Class<?> classType) throws IncompatibleMissionException {  
}
```

Figure 3.3: A sample Mole

The Client API can be used to deploy, run or control Missions; the Molr Server serves the requests made by the Client API; and the Mole Supervisor does the actual execution of the missions on some node in the cluster. The Molr server acts as a proxy for commands and results between the client and the Mole Supervisor. AccTesting is expected to implement the Client API to run/control missions.

3.1.2 Molr use by Developer

A developer uses Molr in two ways: 1) to qualify a class of interest as a mission and 2) to deploy (or submit) the mission to the Molr Server. Figure 3.4 shows how the `Fibonacci` class can be qualified as a mission. This is done by adding a `RunWithMole` annotation instructing the executor to run this class with the `IntegerFunctionMole`

```
@RunWithMole(IntegerFunctionMole.class)  
public class Fibonacci implements Function<Integer,Integer>{
```

Figure 3.4: Qualifying a class as a Mission

Once a class has been qualified, it can be materialized into a Mission object and then deployed to the Molr Server. This service has not actually

been implemented on the Molr server. Nevertheless, this doesn't stop us from defining the types to show the workflow: `MissionMaterializer` (shown in Figure 3.5) could be used to *materialize* a Mission (i.e., create a mission from a qualified class) and the `MissionDeploymentService` (shown in Figure 3.6) could be used to deploy the Mission.

```
public interface MissionMaterializer {
    Mission materialize(Class<?> classType) throws MissionMaterializationException;
}
```

Figure 3.5: Mission materialization interface

```
public interface MissionDeploymentService {
    public void deploy(Mission m);
}
```

Figure 3.6: Mission deployment interface

The current version of Molr server has some pre-initialized Mission objects which can be run on request. The deployment service (classified as a part of the Infrastructure layer) is discussed further in later sections.

3.1.3 Molr use by Client

AccTesting (or any other client) uses the `MissionExecutionService` to request execution of a mission. This can be done in two ways: 1) by running to completion, or 2) by *stepping* through the mission.

```
public interface MissionExecutionService {
    <I,O> CompletableFuture<RunMissionController<O>> runToCompletion(String missionDefnClassName, I args,
        Class<I> cI, Class<O> cO) throws UnsupportedOperationException;

    <I,O> CompletableFuture<StepMissionController<O>> step(String missionDefnClassName, I args);
}
```

Figure 3.7: Mission Execution interface

Figure 3.8 shows the usage of the Mission execution service. The result is wrapped in a future - which can be unwrapped when appropriate. The reason for this is to save the client from waiting for completion of mission execution, as missions may run for a long time.

```

CompletableFuture<RunMissionController<Integer>> futureController =
    mExecService.<Integer, Integer>runToCompletion("cern.molr.sample.Fibonacci",
        42, Integer.class, Integer.class);
try {
    RunMissionController<Integer> controller = futureController.get();
    CompletableFuture<Integer> futureResult = controller.getResult();
    return futureResult.get();
} catch (Exception e) {
    e.printStackTrace();    //logging is for children
    throw e;
}

```

Figure 3.8: Mission Execution sample

3.1.4 Client API & Molr Server

The Client API internally wraps the arguments provided by the client as a request and sends it to the Molr Server. The server - on receiving this request - initiates an execution on a remote Mole supervisor and returns a mission execution identifier, which is then processed and returned appropriately to the client as a `CompletableFuture` object. The Client API offloads the network communication, future handling and (run-time) type safety from the client developer and does most of the implementation under the hood, striving to provide a clean and simple API.

Essentially, the server (almost) acts as a proxy between the client and the Mole supervisor. The primary aim of the server is to provide the client with complete location transparency.

3.1.5 Internals: Remote execution

The remote execution of a mission is facilitated by the Mole supervisor which - being a Spring Boot application - listens continuously for mission execution requests. The server-supervisor communication happens through a REST-like interface. On receiving a request, the server forwards the request (by potentially adding some contextual information) to the mole supervisor (which may have been freshly spawned for this mission).

The Mole supervisor then executes the mission by finding the corresponding Mission class and Mole in its classpath. This implies that the Mission class, the corresponding Mole class, and all their dependencies should be present in the classpath of the supervisor's JVM. The launching of the Mole supervisor is expected to be handled by the Infrastructure layer.

3.1.6 Internals: Input/Output

Figure 3.2 defines a Mole, which shows how input and output types are generic and not restricted by the Molr API. But, at the implementation level, this cannot be the case without actually writing a (de-)serializer for each I/O type. This is because the Client API is expected to return the result in a specific output type i.e., de-serialize the server response. As a result, every input and output type must be clearly defined in the implementation of the Client API. Although this appears to be a bottleneck to integrate new I/O types, this seems inevitable when we want to return a typed response object - which almost makes it look like things are happening locally (complete location transparency).

One way to avoid this bottleneck, is to return a generic serialized format of the response - from the Mole supervisor - as such to the client in some text format. For example, it could be a JSON string/object. But, converting this into an object of the required output type is offloaded to the client, hence creating an additional implementation overhead and reducing the simplicity of usage.

The right trade-off is hard to tell. The current implementation - which is aimed to be used in the context of AccTesting - assumes that the set of all input and output types are well known, and hence prefers the former approach - while not preventing the latter (a generic return type can be added as an escape-route to avoid changes to the Client API implementation).

3.1.7 Internals: Execution Control

The Client API returns a **Controller** in response to a request for execution. If the request is for simply running a mission, a **RunMissionController** is returned, and if it is for stepping through, a **StepMissionController** is returned. The latter can be used by the client to control an executing mission. Both controllers offer cancellation which can be used to abort execution.

These controllers internally call the appropriate server web end-point to control execution of the mission. Currently, the **RunMissionController** has been fully implemented, and the **StepMissionController** is unavailable for use as this can be done only after the server implements the appropriate end points for stepping - which is work in progress.

3.1.8 Internals: Error propagation

On failure, the error is propagated to the client in the form of a stack trace and finally thrown as an exception. This is done by implementing every

step of the communication using the `Try` type. The communication interface of every component returns a value `T` as `Try<T>`. On failure, a `Try<T>` object contains a `Throwable`, and on success it contains the result `T`. `Try` is a monadic type, which is an extension of the more generic `Either<L,R>` monad: `Either<Throwable,T>`.

`CompletableFuture` - which is used extensively in Molr - is also a monadic type. The current implementation takes advantage of the monadic behaviour to compose computations on a value without unwrapping the future (which *contains* the value) at every stage.

```
@RequestMapping(path = "/cancel", method = RequestMethod.POST)
public CompletableFuture<MissionCancelResponse> result(@RequestBody MissionCancelRequest request) {
    try {
        return meGateway.cancel(request.getMissionExecutionId())
            .<MissionCancelResponse>thenApply(MissionCancelResponseSuccess::new)
            .exceptionally(MissionCancelResponseFailure::new);
    } catch (UnknownMissionException e) {
        return CompletableFuture.supplyAsync(() -> new MissionCancelResponseFailure(e));
    }
}
```

Figure 3.9: Mission Cancel REST controller on Server

By using pattern matching, the implementation ensures that the errors don't leak at a specific stage - which could easily happen when propagating errors by checking for them. In effect, this is achieved by offloading a part of the run-time logic to the type system as it appears to be much more reliable for case handling than conditionals. For example, the receiver of a `Try` object is forced to handle both cases appropriately when attempting to get the contained value.

```
@Override
public CompletableFuture<Ack> cancel() {
    MissionCancelRequest cancelRequest = new MissionCancelRequest(missionExecutionId);
    return client.post("/cancel", MissionCancelRequest.class, cancelRequest, MissionCancelResponse.class)
        .thenApply(cancelResponse -> cancelResponse.match(
            (Throwable e) -> {throw new CompletionException(e);},
            (Ack a) -> a
        ));
}
```

Figure 3.10: Mission Cancel implementation on client

With a combination of pattern matching and monadic composition, we are able to reduce the avenues for unhandled error cases.

3.2 Infrastructure layer

The Infrastructure layer requires further thought, requirement analysis and objective formulation before it can be implemented. One of the key open

questions is: How and in what format should a developer deploy a mission?

During primitive investigations towards treating missions as Docker images which are deployed on a Docker cluster using the Docker Swarm API, we discovered that Docker Swarm doesn't appear to be a suitable fit as it does not provide native support for treating executable Docker images as jobs which "run to completion" [3].

Rather, it supports "services" which are non-terminating (infinitely long running) executables like servers, databases etc. One would have to hitch hike their way through the Swarm API for achieving the required kind of Mission execution behaviour, and solve problems such as garbage collecting dead service objects from completed missions etc. This seems unnecessary and a specific hack would immediately break if the Swarm API changes.

Kubernetes, on the other hand, appears to have support for "jobs" which are similar to Missions in a sense that they are executables which are expected to terminate. The other option is to build the Infrastructure layer in-house from scratch in Java. As expected, this would be time consuming. Hence, it is worth exploring existing solutions or tools specialized at this task.

Erlang is a known for solving such distributed problems. To evaluate Erlang, we implemented the requirements of the Infrastructure layer - as an experiment - in Erlang during the CERN Webfest 2017 dubbed under the name FADE [2]. While the experiment was useful in learning about Erlang and its capabilities, FADE requires further work and a better Erlang knowledge before it can be integrated with a project like Molr which has near future requirements of being production ready.

Chapter 4

Conclusion and Future

Molr currently provides a Java framework as a Client API and offers a remote execution service, remote I/O with run-time type safety, remote error propagation, and establishes the foundations for remotely stepping through a mission. The API has been fully designed, and the current implementation servers as a proof of concept to show that it is possible to integrate any generic piece of executable code into the Molr ecosystem and can be made available for execution to a client. This claim is further supported by the fact that we were able to successfully implement a few simple examples of input, output, side-effects and a real use-case for the Beam Interlock System (BIS) of closing the test frequency loop.

While the current version can be used with some manual intervention, more work needs to be done before it can be used to its full potential. The stepping implementation is currently in progress, and needs to be completed. In this project, we only perform primitive investigations towards implementing the infrastructure layer. It needs much more work to enable distributed execution and to offer a flexible, but safe deployment service.

There are plenty of avenues for further development such as adding logging, persisting state of Molr server, developing a monitoring API, providing built-in security measures etc. Given the generic nature and wide application potential of Molr, it has been open sourced [4].

First prototyping confirms Molr to be a viable option for integrating standalone commissioning tests into the operational AccTesting framework. Molr allows for required degree of flexibility and interaction with the executed tests, and to increase the coherency of commissioning campaigns and therefore the overall dependability of the protection systems at the time of machine re-starts.

The next step would be to validate and integrate first commissioning steps via Molr during the next commissioning campaign. In conclusion, Molr helps

further the goal of AccTesting by providing custom integration and execution delegation features, and this project has helped us establish the foundations for a distributed version of Molr while opening many interesting avenues for further development.

Acknowledgments

I would like to primarily thank my supervisors Marc-Antoine Galilee and Jean-Christophe Garnier for their close guidance and support throughout the project. Since the previous version was built by a collaboration of two different groups at CERN, there was plenty of interest, advice and comments from the previous contributors. I was very glad to receive this support from Tiago Martins Ribeiro and Kajetan Fuchsberger. I would also like to thank Markus Zerlauth, Anita Stanisz and Andrej Svec for their comments on the project and helping with the presentation during the last few weeks.

Nachiappan V.
August 2017

References

- [1] The AccTesting Framework: An Extensible Framework for Accelerator Commissioning and Systematic Testing, <http://accelconf.web.cern.ch/AccelConf/ICALEPCS2013/papers/thppc078.pdf>
- [2] FADE - A Framework for Distributed Execution, <https://github.com/fade-cern/>
- [3] Discussion on lack of "job" support in Swarm, <https://github.com/moby/moby/issues/23880>
- [4] Molr, <https://github.com/molr/molr-remote>
- [5] Molr Demo, <https://github.com/molr/molr-demo>