Grid Middleware Development in Large International Projects - Experience and Recommendations

Peter Kunszt
Swiss National Supercomputing Centre CSCS, 6928 Manno, Switzerland
peter.kunszt@cscs.ch

Abstract

The development process in large international projects with multiple partners and different lines of management is not only technically challenging; it is also an organizational and a communication issue. In this paper we summarize the experience in two large European Grid projects, describe the lessons learned, some of the measures taken and finally we give recommendations on what to consider in future large middleware development projects.

1. Introduction

Our experience is summarized based on two large European Grid projects: The EU DataGrid (EDG) [1,2] and the Enabling Grids for E-Science in Europe (EGEE) [4,5].

In EDG, the experience is based on the author’s capacity as the manager of the Data Management Work Package WP2 and in EGEE, on the leadership of the Data Management group in the development activity ‘Joint Research Activity’ (JRA1-DM). The EGEE project was the follow-on project of EDG, so many of the lessons learned have been taken into account. Nevertheless, the efficiency of the development did not increase, but rather decreased in the second project, due to new factors not considered before.

In this paper we summarize the lessons learned in both projects and give recommendations based on the experiences for future distributed development endeavors.

2. EU DataGrid Experience: WP2

The EU DataGrid [1] was a three-year project, aimed at developing and prototyping an international Grid for three scientific domains: Particle Physics, Biomedical Sciences and Earth Observation Sciences. The development was the main activity of the project: there were 5 dedicated work packages for the development of various aspects of the Middleware (workload services, data management, monitoring, cluster management and storage). In addition, there were three work packages dedicated to each of the application domains and three more auxiliary work packages. Also the work packages themselves were distributed efforts across many partners in many countries. The data management work package WP2 [3] that is discussed in this section had over 20 collaborators at any one time, employed by the project mostly only part-time, from 6 countries and even more institutes. The project started as a research project. The collaborators working in WP2 were mostly computer science researchers in the domain of distributed computing, databases or security. Depending on where the people were from, their interests in research varied widely. Most importantly, many people worked on EDG only as a part-time dedication, so depending on their other projects and workload, they delivered their part of the middleware with varied efficiency. The communication overhead was considerable, and with people working sometimes only 20% on EDG, some of the collaborators barely managed to stay involved with the developments. In addition, people changed over time, some joined the project in the middle, others left early, so sometimes expertise was lost, and there were people to be trained constantly.

An additional communication effort was needed to coordinate between the development work packages themselves. For example, there was some overlap between the storage work package and WP2, and also important pieces were needed in the workload management part that related to data management. Since the application work packages were also completely separated from the development work packages, the gathering of requirements and adaptation to applications was done several times, either by direct
interaction with the application domains, through dedicated workshops or simply by estimation.

An additional challenge was the fact that the project has changed its focus in the middle; after one and a half years it changed from a research and development project into a project providing production services. That meant that many of the researchers working on some of the developments suddenly needed to provide support services to the users, which they were neither trained nor motivated to do.

The efficiency of the WP2 effort might be measured to some extent by the level of adoption and the long-term usage of the middleware that was developed in its lifetime. In that respect, WP2 has performed reasonably well, as out of 6 components 4 have been used by the application groups beyond the lifetime of the project, and a fifth was still referred to and extended further by some researchers in computer science. However, WP2 did not do too well in prioritizing the development and to adapt to user needs, especially in terms of integrating the services with other services coming from the other work packages. Unfortunately, also the usage of the developed services and libraries was relatively short lived as none of them were supported beyond one year after the project has finished, so a year later only two of these components was still in use, while two years later all of them have been replaced.

As such, the effort was very useful to build experience and to train people. All of the WP2 members were able to move on and get involved in follow-on projects, but WP2 was not efficient in providing a sustained set of middleware to the Grid user community.

There are several lessons learned that can be summarized from the three years of WP2:

*Part-time development in many very small teams does not work well.* Most of the development work of WP2 has been performed by relatively few people – those who have worked almost full time on the project. The part-time members did not deliver components as promised. Small local teams of 1-2 people working part-time are not productive at all, as these people have nobody to discuss their work with on a daily basis and are prone to losing focus to their other activities. The clear lesson learned is that development should be done in larger (at least 3 people) in dedicated teams, full time if possible, but at least 60% so that in distributed development efforts this is the larger part of a person’s duties.

*Setting down software development conventions upfront is essential.* In EDG there was no uniform software development convention agreed between the different work packages. To agree on such conventions after the development has already started is an additional drain on resources and quite demotivating for everyone involved.

**Common components need a dedicated activity.** Many common libraries relating e.g. to configuration have been developed by more than one work package. To agree later which one to use proved to be one of the main obstacles for integration.

**Some common components (like the security infrastructure) need to be built in from the start, homogeneously in all components.** Some items cannot be ‘added later’ as they need to be built into the each service.

**Applications need to be adapted or built together with the middleware.** EDG simply delivered services as they became available during the project, and applications could only be adapted to them after that. A much closer iterative approach where developers and users interact regularly would have avoided a lot of misunderstandings concerning requirements and would have led to better prioritization. Some serious design flaws due to misunderstandings could have been avoided much earlier and would have led to better, higher quality services.

**Changing focus in the middle of the project should be avoided.** The focus change in EDG had a very negative effect on the motivation of the project members. Many have found themselves doing things that they did not sign up to do and found their main interest relegated to second or third priority, which lowered the motivation especially of the part-time members considerably.

**Middleware documentation has to be one of the largest activities.** The WP2 developments have not all been set up with the users in mind. Many of the documentations have been written as research papers, not as usage documentation for those interested in downloading and installing a service. This was one of the reasons why some of the services have been dropped soon after the project was over.

### 3. EGEE Experience: JRA1-DM

Enabling Grids for E-science in Europe EGEE was a 2-year follow-up project of EDG, where several of the lessons learned discussed above have been addressed, most importantly the production service aspect. Half of the EGEE effort was dedicated to operating issues, and to supporting the users, which was only a very small and hastily added ad-hoc activity of EDG. This has helped enormously to provide production quality services. Also the development has been pooled into one activity of EGEE (JRA1) as
opposed of 5 working groups in EDG. The subdivisions of development teams into workload management, data management and monitoring have been kept. Security has been added as a new dedicated activity (JRA3) and also some development was added as a part of JRA1. Storage and cluster management have been dropped as it was felt that the existing solutions are good enough or that other external products can be used.

At the same time, the development teams have been pulled together, at least in the case of data management, into one place with fewer but full-time people. This has considerably reduced the communication overhead. The applications have also been pulled together into a single EGEE activity (NA4), focusing initially just on physics and biomedical applications. At the same time, a dedicated group was instantiated that would deal with the adaptation of the applications to the Grid middleware provided by the EGEE development activities.

Since security was a big issue after EDG, a dedicated activity was formed just to deal with security aspects. This has worked fairly well on the standardization level, i.e. in finding agreements among countries and partners for certificate management, certificate authority handling and site security policy management. It did work less efficiently on the development side, although now some services for secure user management have been put in place – but not by the development from the security activity but rather from the workload management activity of JRA1 where these services were most needed. JRA1 also had a dedicated integration and a dedicated testing activity, none of which was present in EDG. The integration activity proved to be extremely useful for providing a controlled and homogeneous set of deployable services. Development guidelines have been set up from the start in EGEE and have been controlled by the JRA1 integration team. The testing activity was a little less successful due to the complicated setup of the project deployment process and due to unclear responsibilities between the JRA1 testing team and the service deployment group that had its own testing activity.

For the data management activity of JRA1, having kept a single group of developers at CERN, one could have expected a surge in efficiency after all these points have been addressed. However, the opposite turned out to be true. Applying the same measure as before, i.e. looking at how many of the services provided were used at the end of the project, we can measure a 20% efficiency (1 out of 5) as opposed to 67% (4 out of 6) in EDG. The immediate question is of course: what went wrong?

It turns out that due to the increased size of the project (the JRA1 activity alone of EGEE was almost as large as EDG before) and the additional layers of communication among development, applications and deployment activities, a different, additional communication overhead was generated on the interactivity layer. Inside the activity of JRA1 data management, the communication overhead was reduced to zero, since all people were in one place. But instead of being able to talk directly to the users, there was an additional group now, which also strove to provide the best possible services to the user community. In this setup the worst possible scenario happened: everyone started to develop their own middleware, out of need or out of frustration that their requirements were not properly met. The coordination of these activities has completely failed. By the end of the project, there have been several metadata catalogs, file catalogs, file transfer services and so on developed by the various groups, although only JRA1-DM had a real mandate to do so. And as things go, only one out of 5 developed services was deployed. The other four were not dropped because of their inferiority, indeed they were probably superior to the other developed services but due to their acceptance with the other communities, who have gotten used to their own services already. Also, management was not effective in enforcing the usage of the services that were designed 'properly', and to control and minimize developments due to the 'not invented here' effect. Worse, it was decided that the developments of EDG should be dropped and another middleware (put together by one of the physics experiments) should be used instead – and almost a year later this decision was revoked returning to the initial setting. In terms of lessons learned, the following observations and suggestions can be made:

Establish a continuous, overall system design activity. One of the things that worked extremely well in EGEE was the establishment of a design team that had a representative from each development team, from the deployment group and also from the application and security teams. However, the decisions reached in this group would’ve needed to be properly propagated through all development layers, through testing and deployment by an established, transparent and easy process. Unfortunately, such a process was not completely put in place in EGEE, and some decisions taken by the design team have never been implemented in actual production, or have been deployed only very late.

Decoupling the development from the applications must be avoided. The idea of having an interface activity between the application and the development
teams has completely backfired in EGEE, at least for data management. Both the application and the interface activity itself have started to develop their own middleware services without the knowledge of the dedicated development teams. This happened due to the size of the project and the lack of proper coordination (and control) between the individual groups. Who has what responsibility quickly became also a political issue. In this aspect, the EGEE project was probably simply too large.

*Developments have to be deployed as soon as possible and used in production or pre-production quickly. Deployment activities must avoid developments of their own.* It also happened in EGEE that the deployment activity has developed its own services, as the ‘new and improved’ services from JRA1 were not ready yet. As it happened, these ‘intermediate and temporary’ services became permanent and the ones developed by the dedicated development teams have never been deployed in production. One of the examples is the file catalog developed by the JRA1-DM team: it has never been deployed on the production system, although the temporary solution was delivered at the same time. For other components the reason given why it was never deployed was ‘lack of production experience with the service’. Here the problem could’ve been avoided if the development and testing activities of JRA1 would’ve been better integrated into the production support activities of the deployment teams. The development teams need to be involved in the user support directly from the start to get a fast feedback on the actual system usage. At the same time, if the deployment activity also develops its own services, those services are ill-maintained because there is not enough manpower available to do so properly – those developers are primarily tasked to support users, not to maintain code. In addition, developments outside of JRA1 they did not have to follow the development guidelines, neither the strict requirements described by the applications. So at the end, nobody got what they wanted: a temporary service became permanent, the ‘real’ development driven by requirement went to waste and was never used, and the applications got an ill-maintained service that did not follow their requirements. In the example of the file catalog, this led to some applications developing yet another file catalog service of their own. So EGEE produced by the end of the project four different file catalogs!

*Enforcement of modern development guidelines needs to be assured, including unit testing and documentation.* Although in EGEE there was a defined guideline as opposed to no guidelines in EDG, it has not really been followed by all development teams, and there was no mechanism to enforce it. Especially unit testing and documentation were in very short supply, but there was no possibility to force the development teams to do proper documentation and testing.

*The priorities of the development teams must not be changed too often.* For the JRA1 data management activity it happened several times that the priorities have been radically changed from one month to the next. This was due to the complex management structure of EGEE and the unclear split of responsibility between the management of the development, deployment and application activities. Depending on what ‘important people’ met last, the priorities were changed from one day to the next, resulting in half-finished services, long delays in delivery and poor documentation and testing. Not to mention de-motivated developers.

*The project development, deployment and feedback process needs to be established from the start. This process needs to be as simple and transparent as possible, ultimately with a single body taking the strategic decisions.* This was the root cause of the previous problem. Simply put: in a large project like EGEE with over 100 partners there is a clear risk of too much politics: there were too many generals and officers and not enough soldiers.

### 4. Summary and Recommendations based on Experience

To summarize the lessons learned above, there are a few recommendations that can be formulated based on this 5-year experience.

1. **Plan your project first and stick to it as much as possible**

   Development projects need a preparation lead time, setting down clear priorities and policies. Development practices, coding conventions, joint libraries, etc, need to be defined upfront. And then the policies and agreements have to be kept and enforced throughout the project, with a single decision maker acting as the ‘police’, having real authority to be able to enforce the policies and conventions. Only that person or that body should be allowed to change the policies, agreements or the priorities, and even so it should do everything possible to avoid having to change the plans.

2. **Aim for end to end integration, show results at all stages of the project**
Development activities must be closely coupled with the application people on one end and the deployment people on the other end. This activity must not be micromanaged but it should be set up such that there is a quick feedback loop from the users and the maintainers of the services being developed. ‘Real’ results should be aimed for from day one, the end users need to be involved and shown how they can profit from the joint activity immediately. This helps to always keep a reality check on the developments, assuring that they are indeed synchronized with the needs of the user community.

3. Build a single team

Splitting the development effort into artificial teams based on location (countries) or other political boundaries should be avoided. Everyone involved in the development effort should feel part of a single large team. Implement tight communications even if people are distributed across many institutions, e.g. by establishing mandatory chatrooms, video conferences, setting up processes mandating the maintenance of documentation or internal blogs, wikis, discussion forums, etc. Regular face to face meetings and events, internal seminars also add to the education, team-building and motivation of the development, application and deployment teams.

4. Provide middleware releases regularly, aim for sustainability

Optimally, the middleware developed by a publicly funded project should be made available using the best open source community practices, i.e. having a project page with all documentation, downloads, support lines, etc. Developer documentation is essential if the middleware is to survive the project.

Of course every project depends ultimately on the people who work on it and all of the suggestions here need to be adapted to the individual boundary conditions using common sense.

5. References


