The Role of Configuration Management in Outsourcing and Distributed Development

Lars Bendix
Department of Computer Science
Lund University
Lund, Sweden
Email: bendix@cs.lth.se

Christian Pendleton
Softhouse Consulting AB
Malmö, Sweden
Email: Christian.Pendleton@softhouse.se

Abstract in English—**The use of distributed development teams is becoming more common and for many good reasons. Some of the advantages are that it makes it possible to outsource parts of the development effort, gives access to a larger pool of talents and specialists, facilitates the integration of mergers and acquisitions, and allows for more flexibility in scaling up and down projects. However, distributed development also brings many new problems to be dealt with on a project. It is more complex to manage, tends to create silos between groups, and there is a risk of loss of control over remote groups or people. Traditionally Configuration Management is seen as the infrastructure that allows for the co-ordination of the various activities on a project and it makes sure that work products flow smoothly through different stages of the development process.**

In this paper, we want to investigate to what degree Configuration Management concepts and principles can provide an infrastructure for distributed development teams too and help address some of their special challenges. We first look at what challenges distributed development teams face and categorize them according to how closely related they are to the Configuration Management domain. Then we sketch Configuration Management solutions to some of the related challenges. It turns out that surprisingly many distributed development challenges can be dealt with or made less problematic by simply applying well-known concepts and principles from Configuration Management. Furthermore, a number of other challenges can be alleviated by creative thinking in the implementation of Configuration Management and/or the collaboration between the Configuration Management activity and other activities.

Keywords – distributed development; outsourcing; virtual teams; challenges; configuration management; best practices; categorization; ontology

Abstract in Russian—**Abstract in Russian should be up to 1000 characters in length. For English-speaking authors conference organizers will assist with transiting their abstracts to Russian. (Abstract)**

Keywords-component; formatting; style; styling; insert (key words)

I. INTRODUCTION

The modern globalization means that products are no longer produced close to where they are consumed and modern specialization means that they may even be assembled from pieces produced in different parts of the world. These factors have also had an impact on the way software is produced where the use of globally distributed development teams is becoming more widespread. Early on in the globalization entire development projects were completely outsourced to low-salary countries to bring down the cost of producing software. Handling outsourced software development projects is not without problems and recent years have seen a shift towards projects where only parts are outsourced or projects where teams are put together from the best people without regards to their physical location.

There are many good reasons for being able to work with virtual teams of people that are globally distributed since it gives advantages in various situations. It makes it possible to outsource only parts of a project and to insource it again at a later time if needed. It gives access to a much larger pool of talents and specialists since we are no longer bound to the local supply and, in general, allows companies in "outback" locations to continue to recruit people that may be reluctant to relocate. When companies buy up other companies, the integration of these new business units is facilitated since it is possible to form teams across the limits of these business units. It allows for a much smoother transition between development and maintenance, that in some companies are carried out by separate teams, and, in general, gives more flexibility in scaling up and down teams according to the actual needs of the project at any given time.

On the other hand, distributed development is generally considered to be rather problematic and difficult to handle. The group dynamics are much more complex to manage and most management techniques rely on communication, which is inherently difficult in a distributed setting. It tends to create silos from the various distributed groups, which makes it difficult to cross the borders. There is a perceived risk of losing control over remote groups or people since the manager is not in close contact and, in general, literature reports many challenges of various kinds.

Since its invention in the early 60s, Configuration Management has proven a valuable and indispensable infrastructure to traditional co-located projects. Being Configuration Management experts we were curious to find out what was so special about distributed development projects and what should be the role of Configuration Management in that context. In a previous paper [5], we tried to understand what were the particular challenges posed by distributed development and how these challenges relate to Configuration Management. Doing an extensive literature review we collected a set of 61 challenges in distributed development reported one or more times. We did an initial categorization of these challenges with respect to their relation to the field of Configuration Management. From our analysis it turned out that quite a number of challenges are strongly related to Configuration Management (meaning that it can provide tools and techniques to manage the challenge), quite a few challenges were weakly related to Configuration Management (meaning that it can be part of a solution to the challenge), some challenges were not related at all to Configuration Management - and a surprisingly high number of challenges reported could be categorized as pure negligence of standard Configuration Management concepts and principles.

In this paper, we want to go one step further. Now that we have a basic understanding of the special challenges in distributed development and how they relate to Configuration Management, we want to dig deeper into the categorization and also start sketching solutions for some of those challenges where an implementation of the solution requires something more than ordinary Configuration Management thinking. The problem with our initial categorization [5] was that the granularity of the reported challenges was too varied to be practical and that the sheer number of reported challenges was difficult to manage. Furthermore, our initial focus was to identify and classify the challenges to understand their nature rather than to come up with solutions to actually handle the challenges.

So this next step will focus on simplifying the classification and making more solid the proper relationship that challenges have to Configuration Management, thereby making it simpler to understand and easier to use. Furthermore, this next step will add proposed solutions to a number of challenges where the implementation of a solution is not straightforward and similar to solutions that work in the co-located case. This will make the work more useful in particular to the Configuration Management practitioners that will have to deal with the Configuration Management related challenges. We have established a wide range of different real-life cases that are used to provide a context for our discussion for both the categorization and for the sketched solutions. The direct intended users of our work are on one hand the decision makers that have the power and resources to ask for having proper Configuration Management implemented on a distributed development project and on the other hand the Configuration Management practitioners that will have to carry out the actual implementations. Indirectly we also target all developers involved in distributed development projects, making them know that there are solutions to some of the experienced challenges and who they should ask for fixes.

In the following, we will first present the background for our work. Then we establish the new categorization and sketch best practice solutions to selected challenges. We will then discuss the consequences of our results and suggest further work before we finally draw our conclusions.

II. BACKGROUND

In order to make it easier for the reader to follow the analysis and discussions in the subsequent sections, we will provide some background on the research methodology we have used and also provide a brief introduction to the field of configuration management.

A. Research methodology

Our previous study [5] was carried out as a literature review to establish what distributed development challenges had been reported (our primary sources were [13] and [17]). Originally we had also established a number of cases of distributed development and had intended to use these to carry out a case study to identify challenges and problems. However, it turned out to be more cost-effective to find challenges through literature review and the cases were primarily used to verify and understand challenges. For this next step, where we dig deeper into the nature of the challenges and their relationship to Configuration Management there is little help to get from literature since very little has been published (primarily [7] and [16]). For this in-depth analysis it has been very helpful to be able to put the various challenges in specific contexts for our discussions. Furthermore, we now also focus on providing solutions to the challenges and the specifics about such solutions are often very dependent on a specific context. So to get a broader view of the different possibilities it has been very valuable to have a wide variety of cases to use in our discussions. In the following, we give short descriptions of the cases that have been used as references for the analyses and designs reported in sections 3 and 4.

Case I: Company A is a large development company (>150 developers at this unit), where distributed development is quite common. They hired consultants from company B, which is distributed on two sites within one country with the consultants working from their “home” office. Four development teams at company B are working remotely (three at one site and one team in a separate, all in all about 30 developers). The deliverables from company B consists of source code files. To solve the distribution situation, company A offers a remote desktop connection solution, making the consultants connect from their local offices to terminal servers inside company A. The remote desktop solution gives access to the whole development environment at company including tools for version control, code review, system integration, test etc.

Case II: A big international software company with several offices at many different locations of which four are involved in this case study: two in Sweden, one in Germany and one in India. The main location manages the whole product. All source code is stored in the main repository, and the product is built here. All four sites develop product components and deliver them to the main location for integration with the product builds. There are two kinds of deliveries. One where

the source code is stored in a local repository, and the delivery is a binary component. In the other, source code is delivered and merged with the shared code in the main repository. The product is built on a daily basis at the main location and is tested daily at another location. Build and development tools are the same across the whole project, whereas different sites use different tools for versioning.

Case III: The base of this case is an international company present in many countries around the world. The product in this case has been developed by a team consisting of a dozen people at a site in Ireland and a single person in Latvia. The project has been developed using an agile methodology and the project has used a single shared continuous integration model to avoid any branches. The project has now been ramped up with about 20 people at a site in India. The original group will continue to be in charge of the development, and the India group will be responsible for any consolidation work and bug fixing on the ongoing release. All developers share a common repository, which they access remotely.

Case IV: A division of an international consumer electronics developer spanning several sites across different continents. The major offices are located in Tokyo, Beijing, Lund (Sweden) and San Francisco. The software development organization in this division numbers 1000+ people. Teams are organized around products, components or features as the situation dictates. Product and component responsibility will be located at one site, but development activities can be assigned to teams in other sites. Software development is regularly outsourced and off-the-shelf components can also be included in finished products. The development environment inside the company is very homogeneous with centralized build resources and tool management. For outsourced teams, an SDK is provided in most cases, but if so required, a remote desktop solution is also offered.

Case V: This is a small to medium company that has been successful and grown organically over the years. In recent years it has become increasingly difficult to attract new people to the outback location of the company, which has forced the company to open small branches in 4 major (university) cities. The company tries to keep projects as co-located as possible and in most cases manage to keep projects on one site, maybe with one or two people from other sites participating. However, in some occasions projects have to be manned with 1-3 persons from each of 3-5 different sites.

Case VI (anti-case): This case is the complete opposite of the other cases as it describes an extremely co-located setup where everyone on the project team is in the same room at the same time. The purpose of this case is to uncover CM solutions that are implemented differently on a DD project and CM solutions that are simply "implemented by communication" on a co-located project. A group of 8-10 students has to produce an application to manage motorcycle competitions [11]. They work closely together with a customer, develop following eXtreme Programming and are being coached by two older students who have done the project previously. The team has its own room where all project activities takes place and everyone works in the room at the same time. Each iteration starts with a two-hour planning game and ends with an eight-hour programming session. In between up to four hours of individual work per student can be spent on spikes (e. g. looking through the code for bad smells/missing unit tests, looking into how to use Ant for the release, baking cake - or whatever the team feels they need in preparation for the programming session). The project runs for 6 iterations and the final (and fourth!) release is complete with applications, user manuals, source code, and technical documentation.

B. Configuration Management basics

In this sub-section, we give a short introduction to the most important concepts and principles in CM. It serves as a reminder to the knowledgeable readers and will make it easier for the uninitiated reader to follow our subsequent discussions and reasoning.

Configuration Management

Traditionally CM is considered as consisting of four activities: identification, control, status accounting and audit [14].

The purpose of the *Configuration Identification* activity is to make sure that all the important parts of a project are identified and put under configuration control. These Configuration Items are described and defined and it is decided how the Configuration Items should be named and structured to allow easy retrieval and recognition/identification. Defined groups of Configuration Items can make up configurations (like a requirements specification) and Configuration Items can be related to other Configuration Items to give traceability (like tracing a requirement to its tests and implementation).

In *Configuration Control* focus is on managing changes to configurations. Once a given configuration is stable a Baseline is defined for that configuration. The only way to make changes to a baseline is to create a Change Request (problem report, deviations, waivers are synonyms) and take it through the change management process. The central part of that is the Change Control Board that makes decisions about whether to accept or reject a Change Request based on information provided and that subsequently follows the status of the Change Request through to its closure.

Configuration Status Accounting is the activity that can provide all sorts of information to all sorts of people. Traditionally it is looked at as producing printed paper reports of information about the status of the change management process for the project manager with regular intervals. However, more generally the status accounting activity can make available also more dynamic information (like "who is changing this file") through other types of media (like a wiki) and for other types of "customer" (like testers).

Finally, *Configuration Audit* has the purpose of making sure that we are ready to deliver what has been promised and is done in a more formalized way prior to release. The Functional Configuration Audit is a sanity check on whether the prescribed change management process has been followed – have all the accepted Change Requests gone through all steps of the process to end up in the "closed" state. The Physical Configuration Audit checks whether all the physical parts (like memory card, user manual or help files) of the product are

there and correspond to their description. The Configuration Audit activity can benefit very much from information provided by the Status Accounting activity.

Software Configuration Management

When you are working with products that are digital information it is very easy to obtain a copy of the whole product and work on that instead of just the sub-part(s) that need to be changed. This way of working has many advantages. It becomes possible to test your change or contribution in the proper context, it is sometimes necessary to implement cross-cutting features or bug-fixes, and it creates a stable working environment that protects you against the *Shared Data* problem [2].

However, there are also a number of potential problems with this way of working. As Babich points out, the creation of copies of the product gives rise to the *Double Maintenance* problem where changes or contributions to one copy have to be implemented also in the other copies [2]. This is an error-prone process since changes or contributions may be physically overlapping (modifying the same pieces of information) or logically overlapping (modifying related or dependent pieces of information). The co-ordination and integration of these parallel changes may also give rise to the *Simultaneous Update* problem where part (or whole) of a change or contribution becomes overwritten and disappear [2].

If co-ordination and integration is not always an easy task in a co-located setup where oral communication is possible, it becomes even more difficult for a virtual team in a distributed setting where oral communication is difficult or impossible. Piri et al. document that a number of parameters suffer from distribution amongst others co-ordination. This may possibly be due to a degraded performance on parameters like communication, team trust, mutual support and effectiveness [15].

Version control tools implement a number of concepts and principles that can be used to counter co-ordination problems [8]. The concepts of a workspace and a repository can be used to avoid the problem of Shared Data. The Double Maintenance problem is handled by the merge functionality of the tool and can be made more flexible and powerful if the change set model is supported. The Simultaneous Update problem is more tricky and needs much care. A concurrency checking mechanism makes sure that information in the repository will not be overwritten by mistake in case of physically overlapping changes, but the merge functionality may still overwrite information in the workspace. Long transactions (or atomic commits) make sure that all of the intended change or contribution goes into the repository, thus avoiding inconsistent states in the repository. Finally, strict long transactions ensure that we will not get logically overlapping changes into the repository unless they have been integrated and co-ordinated into the workspace first. Our assumption is that version control tools are good collaboration tools too and used in the right way can make it easier to co-ordinate changes and contributions in the absence of oral communication in a virtual team.

III. CATEGORIZING CHALLENGES

In this section, we will elaborate on the initial categorization that we have made previously in [5]. It served well for our original primary purpose – to find the challenges that exist when dealing with distributed development and to get a first impression of how much configuration management could do to help with those challenges. However, working down into the details and trying to look more at the configuration management solutions to some of those challenges the limitations of this initial categorization became clear. We needed better and more precise working definitions of the relationships to configuration management, we needed to bring down the number of challenges to a manageable number and to make them more homogeneous in size, and we needed to work more on the classification of the resulting challenges as it turned out from our initial analysis and discussions that there were many "grey zones". Here we report on the results of that work.

A. Challenge relationships to Configuration Management

We maintain the four relationships from our previous work, but give more extensive and precise definitions. This will make it more clear what is meant and in turn make it easier to categorize the different challenges.

Strongly related: This type of relationship is characterized by a responsibility from Configuration Management since it regards a challenge that is either completely within the domain of Configuration Management (as defined in section 2) or where the majority of the problem can be solved by applying Configuration Management concepts and principles. This means that for this category of challenges Configuration Management should be the driver in creating a solution and carry out the core part of the work, but may in some cases need some help from other fields. In some cases Configuration Management will have to "think out of the box" since a solution to the challenge in the distributed development case will be different from what would work in the co-located case. The negligence of a strongly related challenge will often have severe consequences on a distributed development project as it is negligence of configuration management.

Weakly related: For this type of relationship, we intend challenges that are not a Configuration Management responsibility and that do not lie within its domain. Since Configuration Management is about providing infrastructure for all types of projects, it will sometimes have data or functionality that is primarily used for Configuration Management purposes, but which might also be useful in other contexts. In such a case some other field is responsible for addressing the challenge and be the active driver, Configuration Management will then act at the passive provider of data or functionality that can be part of a bigger solution. From a Configuration Management point of view, the negligence of a weakly related challenge will not have severe consequences, though it may have so from other perspectives. In some cases it may also happen that Configuration Management activities are affected by a weakly related challenge.

Related, but not particular to distributed development: Challenges in this category have caused us many discussions. On the one hand they are core activities of – or close to – Configuration Management (thus being strongly related), on the other hand we, as Configuration Management practitioners, did not see why they had to be any more a challenge to a distributed development project than to a co-located one. Most of our discussions and clarifications were not so much on how related challenges were to Configuration Management, but rather on whether there was anything in particular for the distributed development case. So in this sense, the alternatives to placing a challenge in this category are not the weakly or not related categories, but the strongly related category (though there may be some border cases). Challenges are put in this category if: the probability or risk that we will experience the challenge does not increase when going from a co-located setup to a distributed development one; the challenge does not get more difficult (or different) to fix and implement when moving from a co-located to a distributed development setting.

This category serves as a "reminder" to everyone (decision makers, Configuration Management practitioners, developers) involved in a distributed development project to always remember to address these "standard Configuration Management issues" not just on co-located projects, but in particular on distributed projects. Apparently some seem to miss out on these issues since they have been reported in literature as challenges – they are not and should not be. It is plain ordinary stuff and does not require you to think out of the box – just to do as you were taught. However, the fact that such issues are indeed reported might indicate that the consequences of ignoring them are more severe on a distributed development project than on a co-located project.

Not related: These are the least interesting from a Configuration Management point of view. The involvement that could be expected from Configuration Management is very little or none at all. Likewise the impact the challenge could have on the practice and implementation of Configuration Management is negligible. This does not remove the fact that challenges belonging to this category might be important and have big effects on a distributed development project. It just means that someone else than Configuration Management will have to be involved in addressing it.

B. Normalization of challenges

From the previous literature review we did, we managed to collect 61 challenges of widely varying granularity. We categorized them into the four different categories above according to their relationship to Configuration Management. However, it soon turned out that there were too many and too heterogeneous challenges for the categorization to be useable and useful. So we decided that we had to refactor the categorization, to unify similar or identical challenges from different sources, to rename (and redefine) a couple of challenges to match better Configuration Management terminology, to remove challenges that were not real and to aggregate smaller challenges into larger, coherent chunks.

Unified challenges: The easy part of the refactoring was to "remove duplets" by identifying challenges that had identical or similar names or meaning. In some cases we had to edit slightly the terminology for brevity and fit. This unification effort gave the following challenges: co-ordination, one SCM environment, communication, collaboration (which in one case was called co-operation), knowledge management, lack of baselines, process support, and risk management.

Other challenges needed a little more editing before they could be unified. Those were "overall visibility" and "group awareness" that were unified to "(virtual) team awareness" which takes care of all aspects of providing information about project status to the team. We unified "project planning" and "project and process management" into "project management" handling all aspects of managing a project from a manager's perspective. Finally, "scope and change management" was edited to "change management" which is a standard configuration management activity.

Removed challenges: We also decided to remove some challenges from the list for various reasons. First of all "software configuration management" as we do not see that as a challenge – rather it is the "tool" that we want to apply to manage the challenges. It may seem like a challenge to get it right to outsiders, but it is no more difficult than any other field in software development. The "time zone differences", "physical distance" and "cultural differences" we see as general characteristics of distributed development and in turn they give rise to the other challenges and are covered by those. We also see "language barriers" as a too general challenge and something that is often also a challenge on co-located projects.

Aggregated challenges: Coming from four different sources, it could be expected that the challenges would have different granularity and that challenges would have overlapping definitions. However, even from the same source challenges would range from very specific and narrowly defined challenges to much more broad-sweeping and fuzzily defined challenges. So we decided to aggregate challenges that we judged to belong to a more general challenge and in some cases split a challenge into two different aspects that were then in turn aggregated to the challenges they belonged to.

The challenge "project management" was extended to include "task allocation", "schedule management", tracking and control" and "quality and measurement" that we consider activities a project manager has to handle when running a project. We included the challenges "tracking and control" and "dispersed software teams do not get information on what other teams are doing" in the definition of the "(virtual) team awareness" challenge to reflect the kind of information that developers could be interested in. For the "change management" challenges, we extended it with "re-plan activities due to scope floating across teams", "the definition of modifications or problems to be handled is unclear", "delay and increased time required to complete change requests" and "change requests are handled at various levels in the project" so it would now reflect all aspects related to the change management activity. The challenges "synchronizing work between distributed sites", "artefacts with different versions and content at each site", "minimizing dependencies between distributed teams" and "dependency" we considered to be different aspects of "co-ordination". The challenge "different

knowledge levels or knowledge transfer" got included in the more general challenge "knowledge management". Finally, "one SCM environment" was extended to include the challenge "project should define one build co-ordinator" and the configuration management tool aspects of the challenges "differences in technologies used" and "tool selection".

C. Resulting categorization

The work we did in restructuring the original challenges to make them more uniform also caused changes in the categorization we had previously made. Some challenges were moved to another category based on a more careful analysis making use of the cases from section 2. However, most of the moves between categories were caused by the definitions of the challenges becoming more precise and explicit – in some cases that would extend the "domain" of the challenge in other cases it would restrict the "domain" and exclude some things that were related to configuration management. Still, we have to point out that there are some border cases where it is not obvious whether a challenge strictly belongs to one category or another.

A: Strongly related: This category of challenges either belong to tasks that are considered core configuration management activities or where configuration management will be able to work as driver and supplier for most of the solution(s) to the challenge.

This category consists of the following challenges (in un-prioritized order):

- co-ordination
- one SCM environment
- communication
- collaboration
- change management
- knowledge management
- (virtual) team awareness

B: Weakly related: The challenges in this category do not belong to the responsibility of configuration management. However, since configuration management collects data and provides infrastructure on projects, it can be part of a solution created by other people.

This category consists of the following challenges (in un-prioritized order):

- project management
- trust

C: Related, but not particular: Much to our surprise as configuration management practitioners, there were a number of challenges that we considered pure negligence of well-known configuration management concepts and principles. Ignoring those would create a challenge to any project, whether distributed or co-located. So these challenges should not really have been there in the first place.

This category consists of the following challenges (in un-prioritized order):

- lack of baselines
- all CIs required for a build should be put under CM
- establish and clarify CM before starting project
- CM engagement in the beginning should be prioritized
- difficult to know the traceability of each module

D: Not related: We did not do very much work on "normalizing" challenges in this category. Apart from the easy unifications ("process support" and "risk management"), once we had established that a challenge was not related to configuration management we did not do any more work on it. The list contains challenges like "need of office space", "different stakeholders", "code ownership" and "process support". It was an interesting and pleasant surprise to us that in the end this category only contained 19 (counting out the unifications) out of the original 61 challenges – less than one third.

IV. Best Practices (for Team Awareness)

In this section, we will present and discuss three different cases. We will both present the case setup, describe the processes and tools used, analyse what kind of problems they ran into, discuss ideas for why they ran into problems and sketch how these problems could be countered. Analysis and discussion of problems and solutions specifically related to the cases will be given here, whereas a more generalized analysis and discussion of problems and solutions related to co-ordination and integration will be given in the next section.

A. Multinational software company

This is a multinational company with development sites around the world (case IV from section 2). The systems are very large and spans over several repositories. The systems are based on hardware platforms and operating systems from different vendors and the repository setup is inherited from the vendors. The company avoids working with virtual teams as much as possible, but the setup sometimes require multi-site teams for certain feature implementations.

The version control is handled with git, a tool that implements long single transactions. But due to the size of the systems and the inheritance from the vendors, this effect is lost when the implementation of a change spans over several repositories. It becomes difficult to keep track of when a change is fully delivered. The solution is to add metadata in the systems for issue tracking and requirements handling. A more severe problem, although less frequent, is to foresee the effect when a change that spans over several repositories is removed from one of the repositories.

Large systems in combination with distributed organization and virtual teams makes it difficult to trace how the systems evolve and the reason for certain changes. In practice, it is solved by "heroes" in the organization taking a major co-ordination responsibility, acting as information hubs where they spend a lot of their time answering questions and helping

others. It makes the projects vulnerable when it comes to resource handling.

Communication about changes often takes place in the code review system which means that the discussion occurs after the code is created. When working in virtual teams across physical distances this, in combination with cultural differences and time zone differences, creates a couple of problems. One is that since you don't know the person that will review your change, you don't want to look bad by making bad looking changes (there are also cultural issues in this area). This gives that developers will want to make their changes complete before submitting to anyone else for review and the changes that are uploaded for review becomes rather big and difficult to review. Another problem is that the solutions are not reviewed until a lot of time has been invested in creating code and developers are therefore reluctant to do major changes to the solution after reviews.

B. Student projects

At Lund University students do a software development project following an agile method at their second year of study at the Department of Computer Science [11] (case VI in section 2). The students are strongly encouraged to work co-located and thus not use their spike-time for programming. Usually students do as they are told, but there is one situation in particular where we have discovered that they "break the rule" – when doing major refactorings. Small refactorings are usually done as part of the stories or tasks, but sometimes they have the need to do a major refactoring. It can be either because they have ignored the small refactorings and built up too much technical debt – or because they have arrived at story 18 which introduces a new type of competition forcing them to change their basic data structure and thus refactor most of their code. We have during the years identified three different ways that our students address major refactorings: the good, the bad, and the ugly.

The ugly way: We often hear horror stories from groups that have "broken" our restriction that no programming should take place outside the Monday sessions. To gain time – and maybe with the good intention of not disturbing the others – some groups decide during the Wednesday planning game that they need a major refactoring of the code base and the task is assigned to a pair to do during the spike-time so there will be a clean, refactored code base to work on on Monday morning. In reality the group now shifts from working co-located in the same time-zone to experiencing all the thrills of time-zone differences. The pair doing the refactoring has no problems. They start from a working code base (compiles, passes unit and acceptance tests) and carefully carry out the refactoring step-by-step making sure that everything still works after each step. And in the end they deliver a code base that works (compiles, passes unit and acceptance tests) and is refactored (either having removed all bad smells or having implemented the new data structure and modified the code accordingly). On Monday morning they happily report that they have done their work and that everything works – and the group starts working. When the first pair does an update – either immediately because they want the refactoring or later when they have finished their task – disaster strikes and they get a lot of merge conflicts because large parts of the code they have changed to implement their task was also changed during the refactoring. Gradually as also other pairs do updates progress slows down and comes to a grinding halt – complete chaos. We have heared of different ways of getting things back on track, but most often what happens is that after some hours of confusion and unsuccessful attempts to solve the merge conflicts, they give up, abandon their messy workspaces and check out a new clean workspace – effectively erasing all the work they had done on their tasks. The worst case of "ugly" is when some of the pairs – maybe in an attempt to solve (or have solved) the merge conflicts – commit to the repository. Then they have to figure out how to return to the "good configuration" they had on Sunday evening just after the refactoring.

The bad way: Some groups adopt the habit of committing their workspaces at the end of each programming session. Maybe because they have heard of the huge merge conflicts from refactorings, maybe because they don't trust their workspaces to be there when they return a week later. In their case they will not experience merge conflicts since the refactoring is not done in parallel with other work. However, they will break the important configuration management concept that each commit should be a coherent and complete logical change. If they commit whatever they have at the end of the day, these commits will often be half-done tasks that do not pass all the unit tests and might not even compile. Our students don't look back, so for them it is not important to identify past logical changes in the repository. However, they will still suffer from the confusion caused by the many half-baked tasks that suddenly appear in their workspace when they check it out on Monday morning. In fact, sometimes pairs doing the refactoring complain that they also have to complete some of these half-done tasks to make sense of their refactoring.

The good way: This year we studied one team that "eliminated" the problem of "time-zone distance" and did it the right way. They designed the refactoring during the spike-time and split it up into 6 smaller tasks, but the actual refactoring coding was done during the Monday lab with all the other pairs present and working in parallel. Whenever a task was tested and verified, it was committed and all other pairs were asked to update and integrate. In all the refactoring took 5 hours to implement and caused no problems. There were two small merge conflicts: one insignificant that was solved in a couple of minutes by the pair that was hit; one that hit another pair and took about 5 minutes of communication between the two pairs to put right (4th and 6th step respectively). Given the ease with which the steps in the refactoring were integrated indicate that at least in this particular case they would also have been able to handle the time-zone differences of a virtual team. Had the 6 tasks of the refactoring been committed to a separate branch, each pair could have integrated them step-by-step when they started working on Monday morning. However, in case of major problems this will not be possible to do without communication and might in some cases cause some of the refactoring steps to be re-done in the light of the resolution of the conflict.

C. *Consulting company*

A consultant company with a mix of on-site assignment and in-house development assignments (case I in section 2). The in-house projects are usually quite short and typically engage 2-5 developers. The teams are co-located but when the project ends, the consultants move on to other assignments. When maintenance work or an upgrade project is started, it is quite common that the allocation is solved with other consultants than in the first project.

Here it becomes difficult to communicate since the work is spread out in time rather than in physical location but the effect when it comes to communication is similar. Since consultants continuously move in and out of projects, it is important that the processes do not rely on physical meetings even though the organization believes in agile methods.

Small organizations tend to document their work less than large ones since it is often easy to get hold of the person that holds some specific knowledge. But when the organization changes heavily over time, documentation becomes essential to maintain knowledge about the products and processes. It is unlikely though, that the developers in a small team feel the urge to document their work for people that may want to read it in the future. Here we can apply the same thoughts as in the student case. Since communication is impossible due to changes in the team constellation over time, the refactoring problem pattern from the student case is present here in a larger scale. The need to be able to follow a chain of changes rather than getting a major "big bang change" can be essential to understand the work of a developer that is not available to explain the change.

V. DISCUSSION

In this section, we widen and generalize the analysis we have done for the single cases in the previous section. We will also discuss how our findings relate to other people's work.

The general lesson from cases B and C is that putting each single, small step as an explicit commit (or change set) is better than having one big commit since it gives the possibility to integrate step-by-step in each distributed workspace. Having to integrate a big change into a mainline that may have undergone big changes too, as in figure 1 (a), will according to the Double Maintenance problem [2] run a high risk of getting serious problems. In fact in case B many student projects have suffered from that experience. In particular when we are dealing with time-zone differences, it becomes important to leave this possibility – and to actually do the integration step-by-step, as in figure 1 (b). Often people are tempted to go directly to the final version on the branch (the end result) and integrate that version since it is already available. This, however, will effectively work as if they were taking in the big commit, as in figure 1 (a) and create the same problems. The right approach would be to integrate changes step-by-step and put possible minor problems right before taking the next step.

One potential problem with this approach in a time-zone difference setup is, that in case it starts to "go wrong" we cannot give feedback to the team working on the branch so they can "change direction" – they have long since done their work and cannot change course. In most cases we should be able to work our way out of it since the compounded change on the branch is supposed to have been tested and work. If not, there is always the possibility to branch the branch and finish the implementation in more synchrony with the task that had problems. This, on the other hand, will create a more complex situation if other people have already co-ordinated with the original branch. So it will be a trade-off what would be the better option.

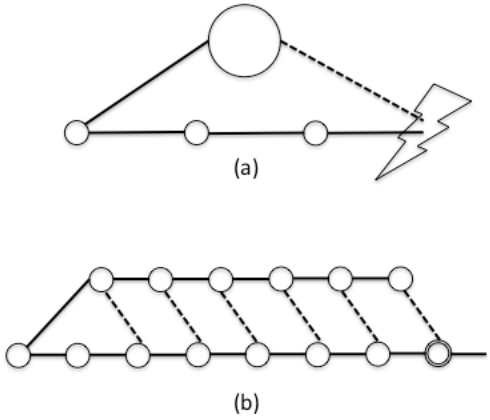

Figure 1. Big vs. small commits

However, an added bonus from doing small commits is, that it allows for much better understanding of what has happened since it can be understood step-by-step instead of having to study only the compounded change. This will become even easier if traceability information is present to other valuable information like commit comments, change request or requirement, or other documentation. This is particularly important in the case of time-zone differences where the possibility of oral communication (dialogue) is difficult or impossible, which to some degree is present also in case A.

A variant on the big commits from figure 1 (a) is that of big integrations. In big and formal organizations you often see a division of labor and responsibility. Programmers are responsible for programming a change, integrators are responsible for integrating changes from programmers and testers are responsible for testing the integration. This setup combined with distributed teams has a tendency to develop into big-bang integration, as in figure 2 (a). Contributions queue up and wait for the integrator to pick them up. The big-bang integration has the same potential problem as the big commits, that it increases the possibility of integration problems. These problems can, to some degree, be countered by being very careful to ensure that all contributions are physically and logically separate. Otherwise, a better approach is to continuously integrate contributions as they become available, as in figure 2 (b), an approach that has proved its worth in the context of agile development methods [3]. Taken to the extreme, contributions are not just integrated continuously but also tested and delivered continuously [12]. In this way, we can

make sure that the "integration bottleneck" is not just shifted a little further down the line, but that each and every contribution is deployable.

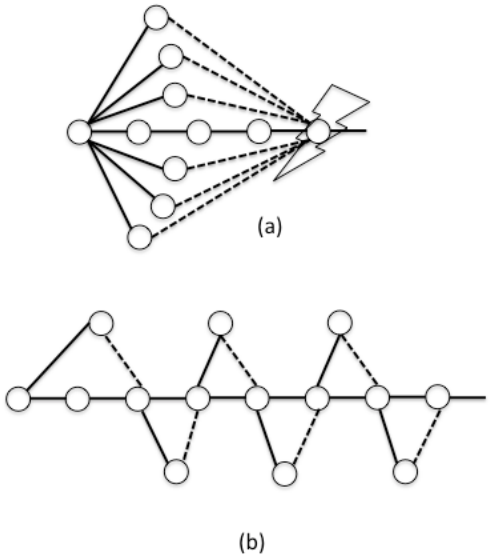

Figure 2. Big-bang vs. continuous integration

Other people have done work that is related to what developers can do to co-ordinate their efforts. Gupta et al. found that people rarely communicate and collaborate in planned ways but most often make ad-hoc decisions [10]. This may indicate the need for a collaboration framework that is based on asynchronous collaboration through a knowledge base. A configuration management database could provide at least part of such a more flexible framework. This configuration management database could also be used to build trust in a virtual team by increasing the information and awareness about what is going on in the project. Piri et al. point out that lack of trust is one of the problems that have to be dealt with on a virtual team in a distributed setup [15].

Alyahya et al. address the problem that distribution makes it hard to create awareness about what changes go on and at what point they are [1]. This affects developers' understanding of development progress. They propose a holistic approach to manage the development progress where they monitor Unit Testing, Acceptance Testing, Continuous Integration and Source Code Versioning. This is quite similar to what happens in the Configuration Control activity after a Change Request has been approved and assigned [14]. They state that it can be a problem that developers can forget to update the status when it has to be done manually, but following traditional configuration management standards such misses will be caught by the Configuration Audit. Finally, the overall awareness they aim at with their proposal is little more than what is (or could be) provided by traditional Configuration Status Accounting [14].

In the case of time-zone differences there was much focus on the usefulness of small commits to allow for easy integration. In the case of physical distance, on the other hand, we are working in parallel with and at the same time as others. Therefore the focus will shift to that of needing awareness of what else is going on and – more importantly – how that might possibly conflict with and influence what we are doing. Traditionally awareness has been limited to what happens in the repository and additional traceability information from the configuration management system since there was no access to personal workspaces. However, with the advent of distributed version control systems we now have the possibility to more closely follow that other people are doing almost in real-time. So we would be interested in continuously doing "virtual merges" [4] to discover physical, syntactical, semantical or logical conflicts as soon as possible. Guimarães et al. propose such real-time integration done in a special merge workspace with the possibility to notify developers of possible problems [9]. Brun et al. carry out "speculative analysis" on all the possible combinations of the state of the repositories of all the developers on a project [6]. Based on this they can supply the developers with information about actual, present conflicts and their severity. This allows developers to act and remove possible conflicts before they grow too big.

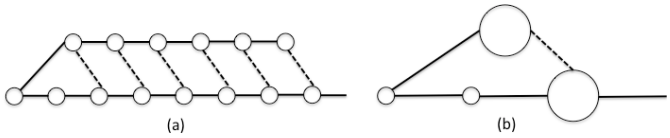

Figure 3. Expanded vs. compressed history

The awareness of what is going on in the project will also be improved by the use of small commits as recommended above. However, a general drawback of the small commits is that the version history becomes too detailed, as seen in figure 3 (a). All this detail is for only one change or contribution and it is easy to see that it does not scale without people loosing "the big picture". It is often seen that the way people try to create overview is by "removing" the intermediate versions. This is not only a violation of one of the most important principles of version control, it also removes the possibility to later understand what was done or to later integrate step-by-step the change to another context. The proper way of dealing with the problem would be for the version control tool to allow to view the version tree at two different levels of detail – one for the detailed description of each single commit (and merge) and one for maintaining an overview of the important results, as seen in figure 3 (b).

VI. CONCLUSIONS

It has surprised us as configuration management experts and practitioners that so many "related, but not particular" challenges have been reported for distributed development. These are things that would bring even a co-located project into troubles if neglected. To us that is an indication that configuration management might be easier to ignore on co-located projects because there are other mechanisms for communication and knowledge exchange.

However, on a distributed project a responsible project manager may use our categorizations to gain knowledge of what challenges he can bring to the configuration manager to

solve – either completely or in part. Likewise, developers on distributed project can look up an experienced challenge on the list and if it happens to be strongly or weakly related to configuration management they could request help in dealing with it.

Configuration managers can use the lists to get a better idea of in which ways they can help out distributed projects. What challenges will be considered their responsibility and what challenges will they be able to help out with. For the solutions to the challenges they will in most cases consist of plain implementations of well-known configuration management concepts and principles, in other cases more creativity is required for the implementation.

From our analysis of the challenges on distributed development projects reported in literature, it turns out that most of those – almost three quarters – are either strongly or weakly related to configuration management and therefore have solutions.

So, distributed development does not have to be all that challenging after all.

ACKNOWLEDGMENT

We would like to thank Jan Magnusson, Sony Mobile Communications, Sweden, Marc Girod, Ericsson, Ireland, Torben Poulsen, Thy:data, Denmark and Ulf Steen, ABB, Sweden for comments, ideas and provision of cases.

REFERENCES

[1] S. Alyahya, W. K. Ivins, W. A. Gray: “Co-ordination Support for Managing Progress of Distributed Agile Projects”, in Proceedings of the First Workshop on Global Software Engineering for Agile Teams, Helsinki, Finland, August 15, 2011.

[2] W. A. Babich: “Software Configuration Management – Coordination for Team Productivity”, Addison-Wesley Publishing Company, 1986.

[3] L. Bendix, T. Ekman: “Software Configuration Management in Agile Development”, in I. G. Stamelos, P. Sfetsos (Eds.) “Agile Software Development Quality Assurance, IGI Global, February 2007.

[4] L. Bendix, P. Emanuelsson: “Requirements for Practical Model Merge - an Industrial Perspective”, in Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems - MODELS '09, Denver, Colorado, October 4-9, 2009.

[5] L. Bendix, J. Magnusson, C. Pendleton: “Configuration Management Stories from the Distributed Software Development Trenches”, in Proceedings of the 7th International Conference on Global Software Engineering, Porto Alegre, Brazil, August 27-30, 2012.

[6] Y. Brun, R. Holmes, M. D. Ernst, D. Notkin: “Proactive Detection of Collaboration Conflicts”, in Proceedings of ESEC/FSE, Budapest, Hungary, September 7-9, 2011.

[7] S. S. M. Fauzi, P. L. Bannerman, and M. Staples: “Software Configuration Management: A Systematic Map”, in Proceedings of the 17th Asia Pacific Software Engineering Conference, Sydney, Australia, November 30 – December 3, 2010.

[8] P. H. Feiler: “Configuration Management Models in Commercial Environments”, Technical Report, CMU/SEI-91-TR-7, Carnegie-Mellon University, Pennsylvania, March 1991.

[9] M. L. Guimarães, A. Rito-Silva: “Towards Real-Time Integration”, in Proceedings of CHASE’10, Cape Town, South Africa, May 2, 2010.

[10] M. Gupta, J. Fernandez. “How Globally Distributed Software Teams Can Improve their Collaboration Effectiveness”, in Proceedings of ICGSE, 2011.

[11] G. Hedin, L. Bendix, B. Magnusson: “Teaching Software Development using Extreme Programming”, in the book "Reflections on the Teaching of Programming" (J. Bennedsen, M. E. Caspersen, M. Kölling (Eds.)), Lecture Notes in Computer Science, Vol. 4821, Springer Verlag, May 2008.

[12] J. Humble, D. Farley: “Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation”, Addison-Wesley Signature Series, August, 2010.

[13] M. Jiménez, M. Piattini, and A. Vizcaíno: “Challenges and Improvements in Distributed Software Development: A Systematic Review”, Advances in Software Engineering, Volume 2009, 2009.

[14] A. Leon: “Software Configuration Management Handbook”, (second edition), Artech House, 2005.

[15] A. Piri, T. Niinimäki: “Does distribution make any difference?”, in Proceedings of the First Workshop on Global Software Engineering for Agile Teams, Helsinki, Finland, August 15, 2011.

[16] L. Pilatti, J. Audy, R. Prikladnicki: “Software Configuration Management over a Global Software Development Environment: Lessons Learned from a Case Study”, in Proceedings of the International Workshop on Global software development for the practitioner, Shanghai, China, May 23, 2006.

[17] F. Q. B. da Silva, C. Costa, A. C. C. França, and R. Prikladnicki: “Challenges and Solutions in Distributed Software Development Project Management: A Systematic Literature Review”, in Proceedings of the 5th International Conference on Global Software Engineering, Princeton, New Jersey, August 23-26, 2010.