Prioritizing SCM for Managing Inconsistency in Distributed Software Project Development

Dillip Kumar Mahapatra, Tanmaya Kumar Das

Abstract : ***The evolution of software engineering has been constant over the past four decades. Some major technological discontinuities, however, can be identified in this progress, which caused a more radical rethinking of the previous established approaches. This, in turn, generated research for new methods, techniques and tools to properly deal with the new challenges.***

Distributed Software Development (DSD) has recently evolved, resulting in an increase in the available literature. Organizations now have a tendency to make greater development efforts in more attractive zones. The main advantage of this lies in a greater availability of human resources in decentralized zones at less cost. There are, however, some disadvantages which are caused by the distance that separates the development teams. Coordination and communication become more difficult as the software components are sourced from different places, thus affecting project organization, project control, and product quality. New processes and tools are consequently necessary.

This paper highlights the software engineering process for distributed software development and related topics in coordination of projects and project artifacts. Different configuration management systems (CMS) approaches and techniques are discussed; these include client-server, k-mutual exclusion, and distributed configuration management systems. New trends in CMS technologies and approaches are also outlined here. Some major areas are addressed in this paper like: how does CMS enable collaborative work; information exchange among clients at different geographical areas and the knowledge management across distributed clients.

Key words : Aggregation, Co-operative, Collaborative, Editors, Knowledge Management, Milestones, SCM, Release, Version, Version-Control.

I. INTRODUCTION

Like every other industry, the SE process is one of the crucial factors bring success to the software manufacturer, it helps all members of the project from the old to the new, in or outside the company can handle the job of their respective positions by way of the company, or at least at the project level. The complexity of managing software development as a key challenge within the field [C.Ebrret & P,2001]. The issue of coordinating the software development process – including the technical side and the human side i.e. the importance of team structure and collaboration among members as key to the success of software development [C.Ebrret & P,2001].

Figure.1 : Software Development Process

Typically a process includes the basic elements like Procedures, Activity Guidelines, Forms / templates, Checklists and tools. We do include some highlighted information on software process;

- *Software Development:* Compliance made by the processes of software development has been defined to produce software products to effectively, accurately and consistently.
- *Project Planning:* Creating the project plan is feasible for the implementation of development and project management software. Develop workload assessment to implementation, establishing appropriate commitments and determine plans for the work that needs done.
- *Software Project Management:* Monitor and review the work already done, the results of the evidence (documents) on the initial estimate, the commitment, planning and adjustment these plans based on the work already done and concrete results.
- *Managing software requirements:* Establish a common understanding between customers and software projects on the customer's requirements will be implemented by software projects.
- *Software configuration management:* Establish and ensure the integrity and safety of the product throughout the project cycle activities of software projects.
- *Software Quality Assurance:* Provide tools and support managers to monitor the processes applied in software projects and software products are built. Review (review) and evaluation (audit) software products and manufacturing operations to ensure that they comply with the procedures and standards specified.
- *Reviewing:* early and effective removal of defective products work (work products). To the development team knows more than the work product and therefore the impairment to be prevented. The products should be considered to be determined during the project and is scheduled in the project plan.
- *Management of products and services:* Construction management policies and service products (SP / DV) to improve the efficiency of the

Revised Manuscript received on December, 2012.

Dillip Kumar Mahapatra, Associate Professor & Head of Deptt., Information Technology, Krupajal Engineering College, Bhubaneswar, Odisha, India.

Tanmaya Kumar Das, Asst. Professor, Dept. of Computer Science & Engineering, C.V.Raman College of Engineering, Bhubaneswar, Odisha, India.

business of the SP / DV. Managing the business organization SP / DV ensure the carrying SP / DV to customers with the highest quality, along with more innovative services. The output of product management policies as rules, procedures, forms together.

In addition to the above, Computer-Aided Software Engineering (CASE) tools are useful in supporting the development of software systems. Upper-CASE tools are primarily focused on supporting the design and analysis phases of software development where as Lower-CASE tools are primarily focused on supporting the implementation and testing phases of software development; [Cheng, L.etal, 2003].

Most software engineering activities involve the coordination and collaboration of documents related to the software system development activities, While many CASE tools exist to help manage the software development process, fundamentally all software engineering activities involve collaborating on documents..

II. DISTRIBUTED DEVELOPMENT SCENARIO

In distributed software development scenario, there is a high degree of parallel development with a potentially high probability that changes made by one user would have an impact on the changes made by another developer. When teams were distributed, the speed of development was delayed as compared to face-to-face teams. Distributed software development also involves four essential problems: evolution (changes must be tracked among many different versions of the software), scale (increased software systems involve more interactions among methods and developers), multiple platforms on which the system will be deployed (coupling the methods and subsystems of the software), and distribution of knowledge (in that many people all are working on the system and each contain a set of the working knowledge of the system) [Chucarroll,2001]. Most importantly, in parallel development , managing the scope of changes within the system over time and to entertain process to manage the software development activities.

In distributed software development, emphasis are to be given from the prospective of *process engineering* (i.e. developers should adhere and comply to some prescribed processes and that they should emphasize communication through documents and through formalized interactions), *information flow(i.e.* analyzing and improving the socio-technical system of a software organization, pointing out the structural problems in its social network) and share understanding(i.e. members of a software organization to negotiate and come to a better *understanding* about their goals, plans, status, and context)

Modern software engineering of large-scale software systems involves a high level of collaboration and coordination. Locasto et al [2002] define three fundamental elements of collaborative systems; all of these are central to software engineering systems, as the core of software engineering is the collaboration and coordination of a project team to develop a system. These fundamental elements are: user management, content management (and version control), and process management. These are defined as:

User management is defined as the *administration of user accounts and associated privileges.* This administration should be as simple as possible to avoid wasted time and confused roles.

Content management is the *process of ensuring the integrity of the data* at the heart of the project. Content management systems often employ *version control* that transparently preserves the progression of the project as the associated documents mature and grow.

A *workflow* is an abstraction of the process that a task takes through a team of people. During the execution of the workflow, it is often difficult and time-consuming to manage individual processes. Process management *handles the interaction between different levels of project contributors."*

In addition to this, CASE tools are being utilized to help and to manage the growing complexity and tightly-coupled activities in distributed software development.

Models of organizational theory suggest that there is a movement away from hierarchical forms of group coordination to utilizing information technology in facilitating more adaptive, flexible structures and offer the possibility for more productive and efficient groups and organizations.

III. CONFIGURATION MANAGEMENT

Software configuration management (SCM) is the task of tracking and controlling changes in the software. Configuration management practices include revision control and the establishment of baselines.

SCM concerns itself with answering the question "Somebody did something, how can one reproduce it?" Often the problem involves not reproducing "it" identically, but with controlled, incremental changes. Answering the question thus becomes a matter of comparing different results and of analyzing their differences. Traditional configuration management typically focused on controlled creation of relatively simple products. Now, implementers of SCM face the challenge of dealing with relatively minor increments under their own control, in the context of the complex system being developed. However, Software Configuration Management deals with "how to control the evolution of a software project".

In distributed software development scenario, there is a high degree of parallel development with a potentially high probability that changes made by one user would have an impact on the changes made by another developer.

When teams were distributed, the speed of development was delayed when compared to face-to-face teams. Also of note was the fact that team members report that they are less likely to received help from distant co-workers, but they themselves do not feel that they provide less help for distributed co-workers. It can be concluded that better interactions are needed to support collaborations at a distance. Better awareness tools such as instant messaging and the "rear view mirror" application offer potential for overcoming some of the problems inherent in distributed software system development.

Clearly, ensuring that users within the shared space have exclusive access to the elements of the shared data and are provided adequate access to the files within the system is of critical importance in distributed software engineering. Configuration management entertains *managing the project artifacts, dealing with version control,* and *coordinating among multiple users.*

A "Distributed Version Control System" (DVCS) is one in which version control and software configuration control is provided across a distributed network of machines. By distributing configuration management across a network of machines, one should see an improvement in reliability (by replicating the file modules across different systems across) and speed (response time). Load balancing can be another benefit of distributed configuration management but there must be *coherent policy* implemented throughout the network.

Enabling distributed configuration management to work efficiently, it is essential that the files/modules are distributed across multiple computers on the network must be transparent to the developer/user. The user should not be responsible to get aware about the fact that where to locate the desired file. Rather, the system should be able to provide an overall hierarchical view of the modules present in the system and the user should be able to find their required module(s).[Magnusson, 1995; Magnusson, 1996].

An interesting aspect of distributed configuration management is the idea that the system provides each user with a public and private space for the file modules [de Souza, 2003]. The public space contains all of the modules in the collaborative, distributed system. The private space contains minor revisions or "what if" development modules that the local user can "toy with" in an exploratory manner; this provides a safe "sandbox" area that each developer can use to explore possible ideas and changes. When a module is ready and needs to be shared by other user(s), it is transferred from the private space into the public space.

Here the major constraint is that the possibility of *deadlock* situation when dealing with specific modules by many users in a collaborative environment, So it is more vital to guarantee that only one user can be editing any section of the collaborative shared space at any given time.

It is more appropriate to choose a policy to deal with this situation is distributed mutual exclusion algorithms.

Three basic approaches for distributed mutual exclusion: one .Token based approach , two. Non-token based approach and three Quorum based approach.

Token-based approach: A unique token is shared among the sites where, a site is allowed to enter its Critical Section if it possesses the token and Mutual exclusion is ensured because the token is unique.

Non-token based approach: Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the Critical Section next.

Quorum based approach: Each site requests permission to execute the Critical Section from a subset of sites (called a quorum) and any two quorums contain a common site. This common site is responsible to make sure that only one request executes the Critical Section at any time.

IV. COLLABORATIVE APPROACH

As a shared set of object modules reside at the central database of a collaborative system, mechanism must be used to coordinate the activities of the multiple users within the system. The more general computer-supported collaborative-work (CSCW) research enables the approaches taken in collaborative software engineering coordination approaches.

Dealing with Releases: We have taken the scenario in the figure below where a shared release (modules) within the software engineering code repository is accessed by two users. If the original version of the release (X) is edited by both users editor 1 and 2, and both users desire to commit their changes to the shared space, then a collision occurs and resolution is necessary. Now the question arises, should editor 1's changes be committed; should editor 2's changes be committed; or should some common version that incorporates both sets of changes be committed?

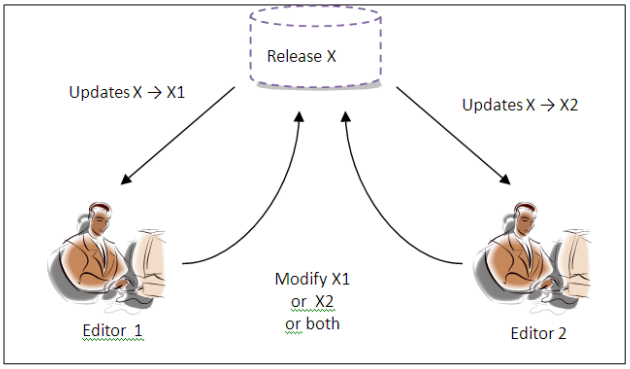

Figure-2: Sharing of Release by Editors

Version Control: In general, benefits of using version control systems include: increasing the potential for parallel and distributed work, improved ability to track and merge changes over time and automating management of revision history. Some specific benefits that can be realized by applying version control to a modeling environment. These are most relevant when models are shared by multiple editors, who may also be geographically dispersed:

- Supports globally distributed model editing by providing a convenient and effective way of replicating models.
- Facilitates collaboration across multiple projects through re-use of common model data.
- Improves performance for widely dispersed teams over slow networks by allowing local storage of models, with global propagation of changes only.
- Promotes orderly changes over chaotic changes and helps to minimize disruptions by separating "work in progress" from "finished" work.
- Helps automate communication within a team by coordinating 'edit' access to controlled information, thereby preventing accidental modification.
- Helps maintain successive revisions of work-to-date with the ability to "undo" changes that are not required, "roll back" to the last "good" version to undo a mistake, or recover from accidental deletions or changes.
- Maintains a work history and audit trail of changes to a model, helping to answer 'Who changed what and when?'
- Team deployment policy: Here we must consider the team deployment policy that also influences version control to answer the questions like;
- Is each member of the team in the same central location and connected via a high-speed network?
- Are the editors likely to work remotely and independently, perhaps disconnected from a shared network for extended periods of time?
- Are there several key locations across the globe where the model will be shared and edited?

To answer the above we must think of the type of team

deployment such as ; c*entralized Team* where all editors are connected via a high-speed network and can therefore share the same physical model hosted in a Database Management

System (DBMS), d*istributed Team* where editors are rarely connected to the same network. They may have to contribute to the model offline, and therefore require a local copy of the model on their own machine and *multiple Site Locations* where Several geographically dispersed sites contribute to the same model. There is no high-speed connection between sites. Teams at each site share access to a local copy of the model.

Coordination: Configuration management systems typically take one of two policies with regard to coordination among clients: optimistic or pessimistic locking. In the optimistic approach, developers are free to develop in a more parallel fashion, but conflict occurs at the merge point when two sets of files must be merged together and changes brought together (and avoid losing work and ensuring that changes in one file have not adversely affected changes in the other file). In the pessimistic approach, developers must obtain a lock on a file before being able to edit it; this can reduce the parallel nature of development since at most one developer can edit the file at any time.

A. PESSIMISTIC APPROACH

The first widely adopted approach to coordination in a collaborative environment is to pessimistically assume that users within the system will desire to edit the same object at the same time and that such edits will be destructive or cause problems. Since this is a shared resource/object, consistency and causality are important. Notice the similarity to causal memory, shared memory, and cache coherency in distributed systems research.

Pessimistic coordination policies are typically implemented using a "check in" and "check out" Application Programming Interface (API). Users may gain access to an unused document by issuing a "check out" request; the document is then locked for that user, and no other user may access the document. When a user has completed any edits to a checked out document, he may issue a "check in" request, returning the document to the repository with any changes made to the local copy.

Since only one user has access to the shared document at any given time, the problem of multiple versions of the same document within the system is avoided. Thus, no two users can have writable copies checked out at the same time. Updates to the repository occur upon a "check in" command, and the old copy of the document is overwritten with the new copy of the document. Often, differentials are saved so that "undo" or "revert to old version" commands are possible. The following figure illustrates this.

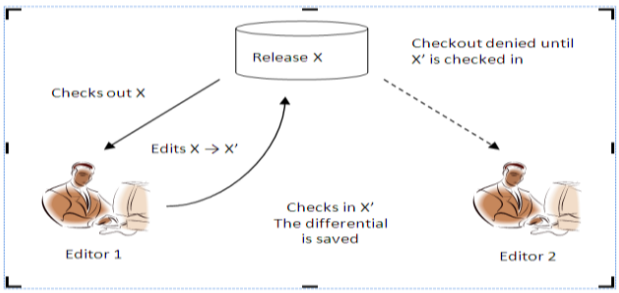

Figure-3: Pessimistic Coordination Policy

One major limitation of the pessimistic coordination policy is the lack of concurrency in the distributed environment; since only one user can access each shared document at a time, then concurrency of collaboration may be inhibited. A few solutions to this problem exist:

First, one can reduce the size of the code placed into each atomic element within the repository. Since each element (document) within the repository contains less code, the probability of two users requesting the same document may be reduced. This is akin to breaking up a large file into smaller files, each of which may be checked out concurrently without being inhibited by the pessimistic locking policy. Of course, it may not always be possible to create small documents within the repository, and a highly-desired document may inhibit concurrency regardless of its size.

Second, configuration management repositories may allow users to check out "read only" copies of an already-checked-out document. I.e. if one user already owns a document, other users may view (but not edit) the contents of this document. Such a local copy could be used within local users' workspaces for "what if" editing without corrupting the original, master copy. If such local changes are deemed relevant to the master copy, the user can later check out the master and incorporate these changes.

B. OPTIMISTIC APPROACH

The second widely used approach to coordinating concurrent development in a shared space is the optimistic approach. This coordination policy assumes optimistically that users will not need to access the same resource at the same time frequently [O'Reilly, 2003], thus this policy promotes increased concurrency among collaboration at the cost of potential problems in inconsistency in the shared documents and loss of causal access. Such a policy is indicative of and seems to work well in an "agile development" environment where communication and productiveness trump tools, processes, and planning [O'Reilly, 2004].

Optimistic coordination systems are typically implemented using awareness within the system such that users are made aware of each others' activities. Awareness is defined as "an informal understanding of the activity of others that provides a context for monitoring and assessing group and individual activities". In such a system, synchronous updates occur immediately when an edit occurs (akin to a write through cache policy in distributed shared memory systems). Consequently, all users have a current copy of any shared document and no check-in and check-out is needed because any document a user is editing is by definition checked out (and perhaps checked out simultaneously by many users) [O'Reilly, 2004]. The following figure illustrates the optimistic coordination policy.

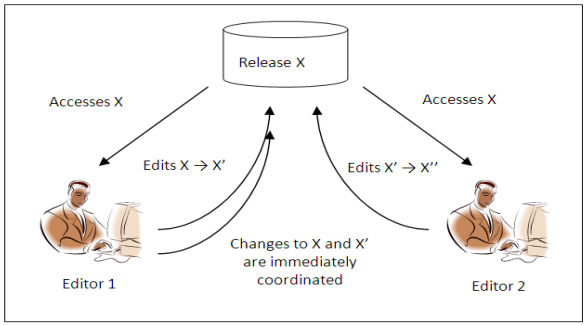

Figure-4: Optimistic Coordination Policy

Such awareness-based optimistic systems rely upon users to coordinate and avoid collisions in edits to the shared document. Consequently, optimistic coordination policies work well in smaller collaborative environments with fewer users when self-coordination is accomplished by the users of the system. The advantage of such an approach is increased collaboration and concurrency.

V. MANAGING INFORMATION

A distributed system consists of a collection of independent geographically dispersed autonomous computer which are connected by a communication network and appears to the users as a single coherent system. Global or centralized controller is absent in this network characterized a distributed system to contain the following five components: a multiplicity of general-purpose resources, a physical distribution of the resources, a high-level operating system, system transparency, and cooperative autonomy. Since the concurrent access of shared resources located at different geographical places is commonly involved in distributed systems, mutual exclusion problem arises frequently in such a system, and the problem is referred as distributed mutual exclusion problem.

It is not too difficult to implement mutually exclusive use of objects if the use of such objects is under a global or centralized controller. However, in a distributed environment, due to the absence of a global controller, the mutual exclusion algorithms become far more complex than those for systems with centralized control. Over the last few decades, several algorithms to achieve mutual exclusion in distributed systems have been proposed,

Solution to the distributed mutual exclusion problem consists of a protocol to be executed among the processes of the distributed system solely by passing messages in order to allow one or some processes to execute private operations with one or several shared resources. The mutual exclusion algorithms can broadly be classified into two families according to their underlying algorithmic principles {permission based algorithms and token based algorithms. The significant algorithms of these two families are discussed is subsequent sections.

Permission based algorithms The permission based algorithms require two (or more) successive rounds of message exchanges among the sites. A site enters into its critical section (CS) only after an assertion becomes true at that site. The assertion has a property that it becomes true only at one site at any time. Therefore, only one site can be in its CS at a particular time. For example; a site has received a REPLY message for its REQUEST message from all other sites.

In this respect,when a process wants to enter a critical section region, it builds the message containing the name of the critical section it wants to enter, its process number, and current time. It then sends a message to all other processes. When a process receives a request message from other processes, one of the following actions are taken.

- If one node receives a request message from other node and if the receiver is neither in the critical section nor interested to enter in it, it sends back an OK message to the sender.
- If the receiver is already in the critical section, it does not replay. Instead, it queues the request.
- If the receiver wants to enter the critical section but has not yet done so, it compares the timestamps. The lowest time-stamp wins. If the time-stamp of incoming message is lower, the receiver sends an OK message. If its own message has lower time-stamp, the receiver queues the incoming request and sends nothing.

If the sender receives an OK message from other member of the network then it can enter the critical section. Therefore, the message requirement per critical section entry is 2(N - 1), where N is the number of network.

Using k-mutual exclusion algorithm in a network of n processes, we allows at most k processes into the critical section (where $1 \leq k \leq n$). The algorithm developed by Walter et al. [2001] utilizes a token-based approach that contains *k* tokens. Tokens are either at a requesting process or a process that is not requesting access to the critical section. In order for this algorithm to work, all non-token requesting processes must be connected to at least one token requesting process. To ensure that tokens do not become "clustered," the algorithm states that if a token is not being used, then it is sent to a processor that is not the processor that granted the token [Walter et al. 2001].

Token based algorithm: token-based and permission-based. In a token-based system, a virtual object, the token, provides the permission to enter into the critical section. Only the process that holds the token is allowed into the critical section. Of interest is how the token is acquired and how it is passed across the network; in some models, the token is passed from process to process, and is only retained by a process if it has need for it (i.e. it wants to enter the critical section). Alternatively, the token can reside with a process until it is requested, and the owner of the token makes the decision as to who to give the token to. Of course, finding the token is potentially problematic depending upon the network topology [Velazquez, 1993].

Distributed token-based, permission-based, and k-mutual exclusion algorithms are all useful in various scenarios. The primary use and impact of distributed mutual exclusion in the context of this paper is to manage access to shared code in the software engineering system. This is a vital part of any distributed software development environment with simultaneous users accessing shared source code.

A. DEALING WITH CONTENTION PROBLEM

The previous sections of this paper have outlined the software engineering processes, software configuration management, and various approaches to coordination. Here we look for dealing with the problem of contention through SCM systems using a technique of locking at various levels of files or documents.

This section will look in detail into the ability for configuration management systems to lock at various granularities. A single file containing many classes and/or methods may no longer be the point of contention among multiple users; now, since this file is broken into multiple sub-files; each of which is managed separately by the SCM system and many users can have parts of the file heretofore large file checked out (or in use) simultaneously.

By adopting sub levels of files for locking, the system may improve concurrency regardless of whether a pessimistic or an optimistic coordination policy is utilized. This is due to the fact that the probability of two users requesting the same document will be reduced when the document sizes are reduced (i.e. if the locking granularity increases, the documents being managed will decrease in size and the number of such documents will increase). Documents in high demand will be partitioned into subsections, and these subsections may be locked independently (with less contention of requests) in a pessimistic system. And in an optimistic system, probability of users editing the same document will be reduced the system may aggregate documents together to form virtual files and other versioned objects from collections of other objects [Chu-Carroll, 2002]. Most SCM systems currently employ the concept of a project (or directory) which is itself an aggregate [Microsoft, 2005a]. Unfortunately, most existing SCM systems do not make it easy to aggregate objects from different projects. But fine-grain SCM systems allow for heightened aggregation as the reusability of each element is increased; if elements are decoupled from other elements, then reuse should increase [Microsoft, 2005]. The following figure demonstrates the aggregation of many elements into larger, versioned objects within the repository.

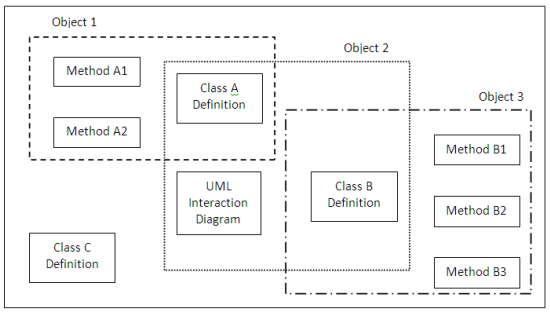

Figure-5: Aggregation Policy

Chu-Carroll et al [2002] propose a model of using fine-grain elements as first-order entities to achieve a high level of aggregation. The idea in their system is that the semantic rules of the programming language(s) in use can guide the automatic management, searching, and merging of various versions of the documents being edited collaboratively; particularly novel in their system, titled "Stellation," is the idea of automatically aggregating specification to implementation, linking code to requirements/specification.

Dealing with complexity: propose an interesting approach to manage the complexity that can emerge when dealing with sub level entities in a SCM system. Since the number of elements in a leveled SCM system can increase by an order of magnitude or more, Magnusson et al implement their system using a hierarchical representation of the code base. Blocks of code contain other blocks of code in a component pattern, and a tree of code blocks is constructed to represent the source in the repository. Depth in the code tree represents such semantic programming language elements as classes, methods, and blocks. To keep the complexity and redundancy of the system minimal, the authors employ a sharing scheme such that references to new/changed entities are "grafted into" the current source tree. The following figure demonstrates the shared source sub-tree from version *n* to version *n*+1.

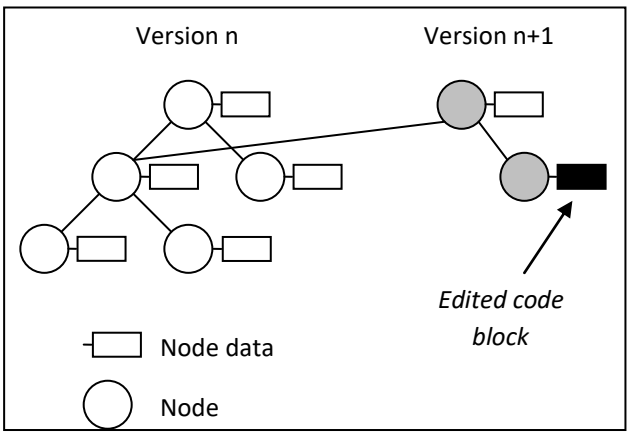

Figure-6: Tree Structure of Shared Resource

Notice in this model only the changed source code tree node data is modified in the structure; the tree node and its ancestors are marked as part of the new version (here the grey nodes shown), but no code is replicated unless it has been modified (notice the black node data above). This avoids redundancy in the source code repository. This model supports additions, edits, single evolution lines (progressions) and alternative revisions (version branches). Additionally, this version tree assists in facilitating merges between multiple disparate edits of a single node.

VI. KNOWLEDGE MANAGEMENT

Distributed Software Development (DSD) allows team members to be located in various remote sites during the software lifecycle, thus making up a network of distant sub-teams. In some cases, these teams may be members of the same organization; in other cases, collaboration or outsourcing involving different organizations may exist. Traditional face-to-face meetings are, therefore, no longer common, and interaction between members requires the use of technology to facilitate communication and coordination.

The team members' experiences, methods, decisions, and skills must be accumulated during the development process through effective information-sharing mechanisms, so that each team member can use the experience of his/her predecessor and the experience of the team accumulated during development, thus saving costs and time by avoiding redundant work. Distributed environments must facilitate knowledge sharing by maintaining a product/process repository focused on well-understood functionality by linking content from sources such as e-mail and online discussions, and sharing metadata information among several tools. Knowledge Management plays important role if organization is also involved in Open Source Software projects, as the quality of the deliverable depend on how well the parallel development on independent software

components across global teams are synchronized.

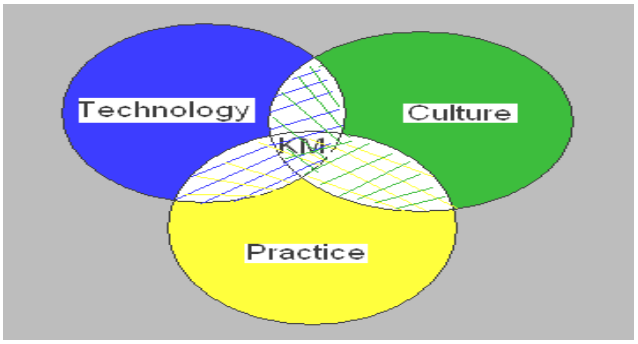

Figure-7: Knowledge Management Paradigm

KM is a combination of technology, culture and practices, management must put equal efforts into all the three aspects for successful implementation. Along with the technology, management should promote a knowledge reuse/sharing culture within the teams, and this must be practiced by making knowledge acquisition process continuous. Without the balance approach to these three aspects, it'll be hard for the management to utilize KM in managing teams effectively. The other important factors for the KM like forming knowledge scout, choosing appropriate platforms, communication channel , the protocols, data access mechanism and expertise to handle technical constraints.

VII. CONCLUSION

This paper has attempted to analyze the history and motivation of software engineering within the context of configuration management in distributed software development field . Issues of coordination and collaboration such distributed software engineering, configuration management, optimistic and pessimistic coordination policies, managing information , knowledge and distributed configuration management have been discussed. While the field of software engineering and configuration management is rich and has a long tradition (dating back to 1968), much work remains to be explored. CASE-based tools are continuously being developed to enhance the productivity of software developers and those who work with them. This last section of this paper examines three interesting fields of future work in collaborative software engineering: the integration of software engineering support into the integrated development environment (IDE), Augur – an interesting software visualization tool, and an open-systems approach to coordinating and managing artifacts among multiple users.

REFERENCES

1. Ebert and P. De Neve, "Surviving global software development," IEEE Software, vol. 18, no. 2, pp. 62–69, 2001. View at Publisher View at Google Scholar.
2. Cheng, L. et al. Jazz: A Collaborative Application Development Environment. In Proceedings of OOPSALA'03, Anaheim, CA, pp. 102-103, 2003.
3. Cheng, L. et al. Building Collaborations into IDEs. ACM Queue, pp.40-50, December/January 2003-2004.
4. Chu-Carroll, Mark C., Wright, James, and Shields, David. Supporting Aggregation in Fine Grain Software Configuration Management. SIGSOFT 2002/FSE-10, pp. 99-108. November 18-22, 2002, Charleston, SC, USA.
5. Froehlich, Jon, and Dourish, Paul. Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. Proceedings of the 26th International Conference on Software Engineering (ICSE'04). IEEE. 2004.
6. Estublier, Jacky. Software configuration management: a roadmap. Proceedings of the Conference on The Future of Software Engineering, p. 279-289, 2000, Limerick Ireland.
7. Gutwin, C., and Greenberg, S. The Importance of Awareness for Team Cognition in Distributed Collaboration. In E. Salas and S. M. Fiore (Editors) *Team Cognition: Understanding the Factors that Drive Process and Performance*, pp. 177-201, Washington:APA Press. 2004.
8. Gutwin, C., Penner, R., and Schneider, K. Group Awareness in Distributed Software Development. In Proceedings of Computer Supported Collaborative Work (CSCW) 2004, Chicago, IL, pp. 72-81, November 6-10, 2004.
9. Locasto, M. et al. CLAY: Synchronous Collaborative Interactive Environment. The Journal of Computing in Small Colleges, vol. 17, issue 6, pp. 278-281, May 2002.
10. Mehra, Akhil, Grundy, John, and Hosking, John. Supporting Collaborative Software Design with Plug-in, Web Services-based Architecture. Workshop on Directions in Software Engineering Environments (WoDiSEE). Proceedings of the 26th International Conference on Software Engineering (ICSE'04). IEEE. 2004.
11. Microsoft Corporation. Visual Source Safe. Available at http://visualsourcesafe.com, February 2005.
12. Microsoft Corporation. Visual Studio 2005 Team System. Available at http://lab.msdn.microsoft.com/ teamsystem/ default. aspx, April 2005.
13. O'Reilly, Ciaran, Morrow, P, and Bustard, D. Improving conflict detection in optimistic concurrency control models. In 11th International Workshop on Software Configuration Management (SCM-11), pp. 191-205, 2003.
14. O'Reilly, Ciaran. A Weakly Constrained Approach to Software Change Coordination. Proceedings of the 26th International Conference on Software Engineering (ICSE'04). IEEE. 2004.
15. Perry, D.E., Siy, H.P., and Votta, L.G. Parallel changes in large-scale software development: An observational case study. ACM Transactions of Software Engineering and Methodology, 10(3):308-337, 2001.
16. Preston, J. A Web-Service-based Collaborative Editing System Architecture. Pending acceptance for the International Conference on Web Services, Orlando, FL, July 2005.
17. Sarma, A., Noroozi, Z., and van der Hoek, A. Palantir: Raising awareness among configuration management workspaces. In Proceedings of the 25th International Conference on Software Engineering, pp. 444-454, 2003.
18. Schneider, Christian, Zündorf, Albert, and Niere Jörg. CoObRa – a small step for development tools to collaborative environments. Workshop on Directions in Software Engineering Environments (WoDiSEE). Proceedings of the 26th International Conference on Software Engineering (ICSE'04). IEEE. 2004.
19. Sommerville, I. Software Engineering 6th Edition. Addison Wesley, Harlow, England. 2001. pp. 4-17.
20. Walter, J. et al. A K-Mutual Exclusion Algorithm for Wireless Ad Hoc Networks. Principles of Mobile Computing '01. Newport, Rhode Island USA. 2001.
21. Wu, D. and Sarma, R. Dynamic Segmentation and Incremental Editing of Boundary Representations in a Collaborative Design Environment. Proceedings of the sixth ACM symposium on Solid Modeling and Applications, Ann Arbor Michigan, pp. 289-300, May 2001.
22. Younas, M. and Iqbal, R. Developing Collaborative Editing Applications using Web Services, Proceedings of the 5th International Workshop on Collaborative Editing, Helsinki, Finland, September 115, 2003.

AUTHORS PROFILE

Dillip Kumar Mahapatra has completed his maser degree in CSE and having more than seven years in teaching UG and PG levels. He has published 15 papers in different journals of national level. He has also authored ten no. text books in the field of CSE and Information technology.

Tanmaya Kumar Das has completed his master degree in CSE and having 22years experience in the field of teaching and industries and having more than 19 papers published in journals of national levels. He has also authored more than 12 no. of books in the field of engineering for UG and PG students.