Distributed data management in product development using Git

André Jordan*
Chair of Information Technologies in Mechanical Engineering,
Otto-von-Guericke University,
Magdeburg, Germany
Fax: +49-391-67-11167
Email: andre.jordan@ovgu.de
*Corresponding author

Klaus Anzengruber and Peter Hehenberger
Institute of Mechatronic Design and Production,
Johannes Kepler University Linz,
Linz, Austria
Fax: +43-732-2468-6542
Email: klaus.anzengruber@jku.at
Email: peter.hehenberger@jku.at

Abstract: Product development takes place in a multidisciplinary environment. Due to this, product data are very different in nature. Furthermore, the manner of how companies collaborate changes. The importance of development networks increases. The changing conditions also affect methods and tools that are used in product data management. In this paper, the relevance of distributed product data management is emphasised. Two existing approaches that introduce P2P technology for PDM systems are mentioned, and possible reasons why they are not used in productive PDM systems up to now are named. Based on this, an alternative distributed PDM approach which focuses on file integration and which uses open source software is presented.

Keywords: product development; distributed data management; PDM; Git.


Biographical notes: André Jordan received his PhD in Mechanical Engineering from Otto-von-Guericke University Magdeburg. He is a research assistant at the Chair of Information Technologies in Mechanical Engineering, Otto-von-Guericke University, Magdeburg, Germany. His research interests include issues related to new forms of work organisation, decentralised product development process and decentralised product data management.
Klaus Anzengruber is pursuing his PhD at the Institute of Mechatronic Design and Production of the Johannes Kepler University Linz (JKU). He received his MSc in Mechatronics from JKU in 2011. He is interested in improved handling of missing and incomplete information in mechatronic design processes with the focus on assumptions.

Peter Hehenberger is an Assistant Professor at the Institute of Mechatronic Design and Production (IMDP) at Johannes Kepler University (JKU) Linz, Austria. He received his Master and Doctorate degrees in Mechatronics from JKU in 2000 and 2004, respectively. His core competencies cover ‘model-based mechatronic system design’, and he has published over 70 peer-reviewed papers and has been guest editor for special issues. Among other scientific service activities, he organises the ‘International Workshop Series on Mechatronic Design’ (Linz, 2012; Paris, 2013; Leuven, 2014). He is also a member of IFIP WG5.1 (‘Global product development for the whole life-cycle’) and the Design Society, where he co-chairs a Special Interest Group (SIG) on Methodologies for Design, Integration, Modelling and Simulation of Cyber Physical Systems.

1 Introduction

Information is very important in product development process. On the one hand a multitude of information is created during the product development process which are needed in later phases and other departments but on the other hand product development operations need information created in downstream divisions to anticipate the solution that should be developed (Vajna, 2014a). Particularly in the early phases of product development, assumptions are used because the needed information is often incomplete or missing (Anzengruber et al., 2014).

Today, a product development process usually takes place in a multidisciplinary environment. Due to this, product data are very different in nature and are related to each other in various ways. By integration, product data should be brought together to form an unitary, complete, consistent, and continuous information basis that allows a non-redundant storage of information and its transfer without interfaces (Meyer et al., 2014).

Information results from data that are put into context and that are synthesised to a higher meaning [Vajna, (p.389), 2014b]. In general, one can find that the data volume is increasing. According to a study of EMC Corporation und IDC Corporate USA, worldwide information doubles every two years (Gantz and Reinsie, 2011). The same is true for product data. Data can change dynamically and are subject to uncertainties. Because of their increasing amount and the growing number of dependencies and interactions between the data, their complexity raises. Tools and methods used for product data management (PDM) have to cope with this complexity.

PDM tools shall guarantee an unitary, complete, consistent and continuous information basis. Companies try to accomplish this by approaches that can be
characterised as centralised. Because of the increasing complexity of product data, the complexity of data structure raise and due to this, the effort to manage all data centrally increases, too.

In this paper, an approach for a distributed data management is presented and discussed. First, an introduction to PDM and PLM is given. The challenges that are caused by the raising complexity are outlined. Based on this, the state of the art concerning distributed PDM is shown. In the subsequent section a Git-based PDM approach is presented. The final section summarises the finding and provides an outlook to future work.

2 Overall context

According to Eigner and Stelzer (2009, p.34) PDM means the management of product and process models, aiming the generation of unique and reproducible product configurations. The basic functions of a PDM system are [Eigner and Stelzer, (2009), p.36]

- project management
- workflow management
- change and release mangement
- viewing, redlining, digital mockup
- input-output management
- publishing
- archive and backup
- data replication
- integrations.

In contrast to PDM, the term product lifecycle management (PLM) is interpreted more broadly. Whereas PDM is primarily used in product planning and product development, PLM covers all phases of the product life cycle. PLM is intended to integrate all product related information, processes, and systems. This includes the business locations and the existing IT infrastructure within, but also suppliers, customers and partners outside the company. In addition to customer needs management (CNM), material sourcing, engineering collaboration and production management, PDM is a part of a PLM solution [Eigner and Stelzer, (2009), p.39]. On the market, there are a lot of providers for PLM systems. Regardless of this, PLM should not be understood as a system that works ‘out of the box’, but rather as a compilation of tools and applications that needs to be individual arranged for each company. The backbone of such a PLM system is normally represented by a PDM system, and thus PDM forms the basis technology within the IT infrastructure of a company [Eigner and Stelzer, (2009), p.37].

The implementation of a PDM system is quite complex. For this reason, PDM vendors provide pre-configured solutions for most industry sectors. Regardless to this, the implementation of a PDM system needs considerable effort to adapt the system
to specific requirements of the company. Therefore, the decision for a suitable PDM solution is fundamental and should be considered thoroughly in advance.

As stated above, PDM systems usually include a workflow management component, to provide the right information at the right time, in right quality and at the right place within the product development process. PDM vendors try to achieve this by merging data in a central manner, and by setting up rules that define which information needs to be provided with which permissions to whom and when. In practice, however, one can note that employees exchange data in a peer-to-peer manner, too. This indicates that the workflow predefined by the PDM system does not correspond to the flow of information that employees consider as convenient.

As stated above, existing approaches aim to manage all product life cycle data centrally. The more data should managed and get linked to each other by PDM system, the more complexity raises. This causes issues eg. in ensuring data integrity and workflow management. Although PDM systems normally own a workflow component, indicators suggest that the product data model used in a common PDM system does not represent its usage within the product development process. Against this background, in this paper an approach for a distributed PDM is be presented. As PDM systems offer a wide range of functions, in this work focus is led on data versioning, replication, and sharing.

3 Distributed PDM

3.1 General remarks on distribution, decentralisation and federation

The dictionary Merriam-Webster.com (2016) defines decentralisation as “the dispersion or distribution of functions and powers; specifically: the delegation of power from a central authority to regional and local authorities”. The term is also used in systems theory. It describes a fundamental principle of complex adaptive systems which means that some parts or functions of a system are controlled in a distributed way. The nodes (or agents) of a decentralised system are autonomous in a certain extent. They are able to get connected to each other to send and receive information. Decentralisation is a condition to form self-organised networks (Johnson, 1999).

Decentralised systems originate in a top-down manner. In contrast, a federated system is originated through the union of individual systems that keep their autonomy. And thus, a federated system is formed bottom-up. Both approaches result in a distributed system. Whereas there is no common definition, a distributed system typically satisfies the following criteria [Ghosh, (2015), pp.3–4]:

- multiple processes
- interprocess communication
- disjoint address spaces
- collective goal.

Distributed systems can be described by their degree of connectivity. Highly connected systems form a mesh network structure that allow a direct flow of information between their nodes (Johnson, 1999). Mesh networks own considerable advantages compared to
Hierarchical structures. They are very robust against disturbances. If some nodes fail, information will be routed to its destination in a different way. Mesh network’s provide ‘self-healing’ capabilities and can balance load between the nodes. They are scalable, which means new nodes can join the network to handle growing demands. In addition to that, data can be stored redundantly, so even if one of the data sources is unavailable, the information is still present within the network.

Against this background, distributed systems have a growing importance in computer science. Ghosh (2015, pp.4–5) points out the following reasons:

- geographically distributed environment
- speed up
- resource sharing
- fault tolerance.

If all network nodes of a distributed system are considered as equal, the architecture is called a peer-to-peer (P2P) structure. In recent years, a lot of research and development work has been done in this area, so there is a wide variety of concepts and tools developed to support P2P technologies.

P2P is often associated with file sharing and illegal downloading. However, P2P approaches are suitable to support searching and distributing information, and thus they have potential to be used in business applications. Regardless of this, applications based on a peer-to-peer architecture have not been used in business environment. The only exceptions are some messaging and video conferencing tools (e.g. Skype), which use a semi-decentralised architecture. Gupta and Awasthi (2009, p.424) point out the following reasons, why businesses and large organisations have stayed away from deploying P2P applications:

- risk of data exposure
- risk of security attacks
- risk to organisational compute resources (when allowing outside access: trust results that are computed by other peers)
- risk of degradation of network performance (due to P2P traffic overload)
- risk of loss of employee productivity.

Gupta and Awasthi (2009, p.425), propose mechanisms to ensure security and to enable companies to deploy cross-organisational P2P applications.

### 3.2 Approaches and examples for a distributed PDM system

Regarding data management, a distributed PDM system manages documents and data locally. In a centralised data management system, network nodes are typically structured in a hierarchical way: Many data generating nodes are faced to one or very few data storing nodes (see Figure 1). In contrast, nodes of a distributed data management system are able to store data for themselves. They can act as data source and data sink at the same time. In this setting, participants of the product development process become
network peers, which are able to get directly connected. Figure 2 shows a fully-mesh network of PDM peers, where all nodes are connected together. A centralised PDM system is no longer required, because information is transferred from node to node until it reaches its destination. Furthermore, Figure 2 points out that product data should be subordinated to the development process. Since data enable and support process instead of ruling it, changes on the process level can be more easily realised.

**Figure 1** Scheme of a common PDM architecture

![Common PDM Architecture](image1)

**Figure 2** Scheme of a peer-2-peer PDM architecture

![Peer-2-Peer PDM Architecture](image2)

One of the issues of distributed networks is that the knowledge of a process is local (Ghosh, 2015, p.8). Because there is no node ordinarily expected to have global knowledge, information stored within a distributed PDM system is some kind of fuzzy. It may contain contradictory data. One and the same request can yield different results depending on which nodes are available and how they are connected (network topology). This leads to questions with regard to reproducibility and traceability of product data, two basic principles of product development that are not called into question up to now.

Furthermore, contradictory data endanger the integrity of data. Central PDM approaches define their own data structure and provide powerful mechanisms to keep the data consistent. In a distributed PDM system, the product data model is fragmented into pieces that are managed by the PDM peers. Although there are mechanisms to ensure data integrity in distributed systems, too, temporary inconsistency is often inevitable here (Bright and Chandy, 2003; Mittal et al., 2015).

Handling fuzzy information is not a specific problem of distributed data management (Lorenz, 2008; Suss and Thomson, 2012; Thunnissen, 2004, 2005). Uncertainties appear in many different ways during the product development process and must be managed in centralised PDM approaches, too. One possible solution is to add properties like reliability and validity to every data set that can be used for evaluation. Destercke et al. (2013) describe a method to handle uncertainty in a data warehouse. A similar approach could help to handle contradictory data in distributed systems.

Conrad (1997, p.169) states that consistency mechanisms can be realised in different ways. First of all, one can define a common protocol that is used by all tools that act as process participants. Another approach is based on wrappers, that link the nodes.
Distributed data management in product development using Git

The protocol-based approach requires a modification of the software tools used by process participants. They need a component for communication and negotiation with other nodes and a constraint manager to monitor data integrity and to handle integrity violation. Usually, software modifications can only be done by software manufacturers, so a common agreement of all software manufacturers about standards is needed. Although there are standardisation initiatives such as Code of PLM Openness (CPO)\(^1\), it seems unlikely that generally accepted PLM standards can be established, due to variety of authoring systems and the dynamics in software development.

In contrast, authoring tools are left unchanged in the second approach. Additional functions such as communication or data storage are provided by wrappers. A wrapper provides an interface to its corresponding authoring tool but also to other wrappers. In this scenario, wrappers are network nodes, and authoring tools are linked indirectly via their wrappers.

There are already approaches known that propose the use of P2P technology for PDM systems (e.g. Hayka et al., 2003; Schlingensiepen, 2008; Stiefel, 2011). Potential uses are seen especially for SME and development cooperations. Schlingensiepen (2008) describes a corporate PDM system that is built up from known technologies and methods, this includes grid technology, internet protocol (IP), independent and exchangeable software modules (business objects), smart clients, software agents, cryptography, classification systems and distributed systems. In his approach, product data are stored using P2P business objects that are interconnected via a distributed hash table (DHT) (cf. Garces-Erice et al., 2004; Zhang et al., 2013). Each P2P business object is bound to an ‘anker node’ that is responsible for the data and that manages the relationships to other P2P business objects. DHT and anker nodes are also used to implement versioning of P2P business objects in distributed networks. Furthermore, Schlingensiepen also proposes solutions for authenticity, permissions, and user administration services.

A similar approach is pursued by Stiefel (2011). He describes a multilevel architecture. On the backend, FreePastry (Hoye, 2009) is used as P2P overlay. The middleware level which is named Decentralised Ressourcen Management (DRM) consists of three sublayer. The first layer CollabNetwork encapsulates the basis operations of FreePastry and provides generic methods which are necessary to establish a decentral collaboration network. The second layer ProjectNetwork encapsulates operations of CollabNetwork and provides basic methods which are specific for product development. These methods are provided as services by the third layer SOAProjectNetwork [Stiefel, (2011), p.209].

One can note that the approaches mentioned above are not used in productive PDM systems up to now. Although, the needed IT technology is available and prototypically implemented, only single aspects flow into the development of PDM systems. The approaches include concepts known form federated database systems which can be considered as state of the art (cf. Conrad, 1997; Härder and Hergula, 2002). Federated database systems unite several heterogeneous databases to one virtual homogeneous system. For this, it is necessary to adopt about a common database scheme. Database integration can be generally done by approaches called local-as-view (global scheme keeps unchanged, when nodes are added or removed) or global-as-view (local schemata impact the global one) (Kensche et al., 2007). As both approaches require changes of the database schemata, provider-independent federated database solutions are difficult to realise.
Experience shows, that SME avoid investing in high level PDM systems but using file servers instead. For this reason, the distributed data management approach described in this paper focuses – in contrast to the approaches mentioned above, which are based on databases – on file integration. The approach is built up on Git, an open source application that is used in software development for change and release management (https://git-scm.com).

4 Git-based PDM approach

In software development, change and release management is undertaken by a version control system (VCS). It is used to record changes to a file or a set of files over time so that one can recall specific versions later. Thus, the functionality offered by VCS and PDM are partially similar. Git is a distributed version control system (Chacon, 2009). The software is free and open-source (F/OSS) (cf. Feller et al., 2007). In the following, a short overview about Git will be given. After this, a concept of a distributed PDM system based on Git will be presented.

4.1 Overview of Git

As mentioned, Git is a distributed version control system. It can either be used as a standalone solution or in a collaborative manner. In Git, all files are stored locally within a repository. Changes are stored as snapshots of the current directory, which are created using the commit command. Besides the snapshots, commits also store the ID of the preceding commit. By this, the commits form an unambiguous chain, that records all changes.

Although Git was originally designed for the Linux kernel project, it is not limited to software development. Git is fast and efficient even in large projects, and, because of its branching concept, it is suitable for non-linear development projects. With Git, workflows can realise many different ways. Chacon (2009, p.97) describes three variants to use Git in a distributed manner:

- **Centralised workflow** – All contributors use a shared repository to publish their work. This workflow is preferred in small teams.
- **Integration manager workflow** – In this scenario, contributors have write access to their own public repository and read access to non-owned repositories. Teams using this workflow may define an official repository that is maintained by an integration manager.
- **Dictator and lieutenants workflow** – This kind of workflows can be used in very big projects or in highly hierarchical environments. In this setting, there is more than one level of integration. Contributors base their work on the official repository that is maintained by a project leader (‘benevolent dictator’). Their contributions go to the repository of the integration manager (‘lieutenant’), who is in charge of them. The project leader takes the changes of integration manager’s repositories and merges them into the official one.
For both, integration manager workflow and dictator and lieutenants workflow, contributors are advised to use a private and a public repository. Whereas the private repository is local and offline, if it is not in use, the public repository is stored on server, so that the files are available even if the contributor switches his computer off.

The Git workflows described above are implemented in a hierarchical manner. They are used by the most software development projects. Nevertheless, Git also supports a pure P2P workflow, which will be described later.

As stated above, Git is known for its branching concept. In software development, a branch represents an alternative development line. Branches are used to implement a new feature or for test purposes. New branches can be either created based on the current snapshot or on a precursor snapshot. Merging two branches creates a snapshot. Figure 3 shows the process of branching and merging.

**Figure 3** Branching and merging

In Figure 3, there are two development branches that are based on snapshot A of the master branch. The developer works on features X and Y independently and creates a sequence of snapshots for both of them (snapshots B to C and D to E). By merging, the results of the development branches are integrated into the master branch: first, snapshots A and C are merged to form snapshot F, then snapshot E and F are merged and create snapshot G. The changes that are made are combined in way that the resulting data are consistent.

A different way to merge branches is called rebasing, which is shown in Figure 4. In contrast to the merging strategy described above, rebasing does not create a new snapshot. As Figure 4 shows, changes that are made in snapshot B and C are directly applied on snapshot E which means that B and C are appended to E.

Besides merging branches, Git also allows to add a selected snapshot to a branch. For this, Git provides the cherry-pick command. As Figure 5 shows, the master branch contains the snapshots A and B, whereas the snapshots V, W, X, Y, and Z are committed on a different branch. By cherry-picking, the snapshot X is included into the master branch. Based on X, the snapshots C and D are committed.

In contrast to most other version control systems, branch management is implemented in Git in a lightweight manner and encourage developers to create and use branches often [Chacon, (2009), p.44]. Contributors can use branches as ‘playground’.
If the status of their development is ready to publish, they merge their changes into the master branch, so that other contributors or integration managers are able to access them.

**Figure 4**  Rebasing

![Figure 4](image)

**Figure 5**  Cherry-picking

![Figure 5](image)

To enable collaboration, repositories need to be linked together. For this, Git provides on the one side the `clone` command, which can be used at the beginning of a project to clone an existing repository, and on the other side the `remote` command, which can be used later in the project to add a remote repository to the local one either entirely or selected branches only. By this, a developer is able to access commits of other contributors and use them for his own work. To transfer changes between repositories, the commands `pull` and `push` can be used. The `pull` command is used to fetch changes, that are made in a remote repository, and merge them into the local one. In contrast, `push` is used to integrate local changes into a remote repository. For this, the pushing peer requires write access to the remote repository.

Git was developed with open source software projects in mind. Most of these projects are covered by a free software license, which allows the user to modify and to redistribute the software (Feller et al., 2007). For this reason, management of user permission is not the main focus of the Git development. However, there are many use cases where user permission must be taken into consideration. Innately, Git supports a basic user management via SSH. For this, it is necessary to create a user on the server and append the public SSH keys of the approved contributors to the `.ssh/authorized_keys` file. By this, access to all Git repositories created on the server can be restricted to registered users. If fine-grained permissions are required, Gitolite (https://github.com/sitaramc/gitolite) might be a solution. Gitolite is a wrapper around a base Git installation that facilitates the secure management of repositories and of the user privileges governing access to those repositories.
In addition to this, push policies that describe guidelines, permissions, and limitations can be enforced in Git by ‘hooks’ [Chacon, (2009), p.178]. In software applications, hooks refer to methods to extend the functionality of a software. Hooking is often based on intercepting function calls, messages or events passed between software components.

It was mentioned that developers can add remote repositories to their local Git repositories using the remote command. By this, the remote repository is closed integrated to the local one. Changes, which occur in the remote repository, are recorded by the local commit history. Besides this, Git owns a concept named submodules. By this, remote repositories appear as subdirectories in the local Git repository. The commits of a submodule are kept separate. Submodules are used when the data managed by the repository should be developed independently from each other. In software development, integrating a library into a project is a typical use case. Submodules can be used to structure a complex project and create a deep or a flat hierarchy depending on the requirements.

4.2 Concept description

In the course of product life cycle, nearly a vast number of different data are processed by a multitude of various applications. Due to this, integrating the data by using a common database scheme seems almost hopeless. For this reason, the approach described here is more ‘low level’. Instead of a seamless integration of the data within the process, this approach focuses on making the data available in the first place. Thus, the approach is based on file integration.

Furthermore, it has become apparent that the activities in product development can be described as a network. The participants of the product development process should be able to act flexible in case of unforeseen circumstances or disturbances. For this reason, they should be able to get connected to each other directly and exchange data and information. In addition, the resulting network needs to be flexible enough to expand by connecting new nodes, but it also should be able to remove connections which are obsolete. Against this background, the PDM approach should support file transfer in a P2P manner.

As described in Section 3.2, distributed PDM approaches already exist. Although the approaches make use of known technologies, they are not practically applied. A reason for this might be that the development of an application needs great efforts which can only be accomplished by large providers. Since a PDM approach should cover a wide range of applications that are used within the product development process, a provider independent solution is intended to be developed. It can be reasonably assumed that PDM providers have neither a significant incentive nor the possibilities to develop such a PDM solution. For this reason, the prototypical implementation described here is based on open source software and open protocols. By this, the number of possible contributors increases. In principle, each stakeholder can contribute to the solution, adopt the application to its needs and develop add-ons.
As shown above, Git is suitable for distributed file management. As the tool is mainly used in software development, in the following, a prototypical implementation should be described to show how Git can be used in the area of product development. In particular, some specifics will be mentioned.

Basically, the approach provides that a Git repository is created for each process participant. Branches are not only used to represent an alternative line of development but also to provide files to other process participants (see Figure 6).

Figure 6 Providing files to other process participants using transfer branches

As Figure 6 shows, a set of files is created or changed during the development process. A commit is made for each file in the master branch of the process participant \( P_1 \), and by this, snapshots are created. To provide the files to other process participants \( (P_2, \ldots, P_n) \), the snapshots are merged into the corresponding transfer branch using the cherry-pick command. In Figure 6, for instance, the files fileA.cfg, fileB.stl and fileD.doc are provided to process participant \( P_3 \). \( P_3 \) can include this transfer branch into its Git repository using the submodule command. By doing this, \( P_3 \) can use the files for its own work, merge the snapshots to its master branch (see Figure 7), and provide the resulting files again to other process participants by defining transfer branches. Since the access to a branch can be restricted using Gitolite, process participants can only read those branches for which they are authorised.

As stated above, in general, branches represent alternative lines of development. In Git, a branch is based on a commit and thus, it contains all preceded commits, too. Since the commit history of the transfer branches should be empty, when they are created, transfer branches should base on the very first commit. In Figure 6, this initial commit is labelled "0". It contains a file with some general information (e.g. about the owner of the repository) that can be given to all process participants.
Figure 7 Providing files from other process participants using transfer branches

Having regard to the above, process participants negotiate with each other about who needs a file, who can provide it and in which file format. The agreement is based on the intended workflow. Since process participants are able to create connections to others participants and remove those which are no longer needed, it is ensured that only those files are provided which are necessary for the workflow. On the other hand, process participants can optimise the workflow by organising the data in a appropriate way. As the workflow can be influenced by e.g. changing the file content, the file format or the providing date, one can state that the workflow is defined by the process participants in a self-organised manner.

As mentioned, Git has its origin in software development. In this area, repositories primarily contain source code files which can be viewed and edited on text terminals or text editors.

The tools that Git uses to identify differences between two versions of a file or to apply merge strategies are optimised for text files.

In product development, there are CAE solutions that use plain text files, too, to exchange data and to control the application. Software products like MSC.Nastran, LS-Dyna or MatLab are examples for this. As their file format is open and well documented, the tools can easily integrated in existing processes and workflows. By this, users are able to create batch processes by concatenating different applications via their input and output files (Vajna et al., 2006). Because of the plain text format, those files are easy to exchange and to version-control via Git.

However, there are also a lot of applications in product development that use a native binary file format. Modern CAD/CAE solutions, such as CATIA, Siemens NX or PTC Creo, are examples of this. Their files are often much larger than source code files.

Due to the closed file format, version control using Git is hampered because the structure of the file must be known for the merging process to generate a valid output. However, Git is able to handle binary files. When changing a binary file, a copy is created by Git and stored in the repository. Due to this, a Git repository grows by the size of the file with each new version. Thus, most benefits of Git will not come to fruition. To solve this issue, user defined diff and merging tools can be assigned to Git. CAD systems as Siemens NX, CATIA, and PTX Creo are able to compare two versions
A. Jordan et al.

of a model and present deviations in graphic and tabular form. Such tools should be further developed as they have the potential to perform diff and merge function in a Git-based PDM system.

The PDM approach discussed here provides only fetching of selected branches instead of cloning entire repositories. By this, the necessary hard disk capacity which is needed to store the repositories can be reduced. Regardless of this, versioning of large files may still occur problems, since Git itself does not provide any special approach to handle large files. To solve this issue, an extension named Git large file storage (LFS) was developed by GitHub (https://git-lfs.github.com/). Git LFS replaces large files such as audio samples, videos, datasets, and other binary assets with text pointers inside Git, while storing the file contents on a separate LFS server. By this, checking out branches from a remote server can be accelerated because large files are only downloaded when they are needed.

4.3 Comparison with other approaches

As described above, the distributed PDM approach based on Git basically provides files to those stakeholder who need it. One may ask, that the differences are to other file-sharing applications e.g. Dropbox or ownCloud. Both applications are web-based and follow a client-server-approach. In Table 1, some selected characteristics concerning version control and collaboration are compared.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Dropbox</th>
<th>ownCloud</th>
<th>Git</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version control</td>
<td>file-based*</td>
<td>file-based</td>
<td>commit, tree, blob</td>
</tr>
<tr>
<td>Version size</td>
<td>file snapshot*</td>
<td>file snapshot</td>
<td>packed diff file</td>
</tr>
<tr>
<td>Version history</td>
<td>basic: last 30 days</td>
<td>complete</td>
<td>complete</td>
</tr>
<tr>
<td></td>
<td>pro: last year</td>
<td>business: complete</td>
<td></td>
</tr>
<tr>
<td>Sharing</td>
<td>file/directory</td>
<td>file/directory</td>
<td>branch/repository</td>
</tr>
<tr>
<td>Conflict handling</td>
<td>file copy</td>
<td>file copy</td>
<td>auto merging, manual merging</td>
</tr>
<tr>
<td>Access</td>
<td>web gui, desktop client, webdav</td>
<td>web gui, desktop client, webdav</td>
<td>web gui, desktop client, webdav</td>
</tr>
<tr>
<td>Federation</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Licence</td>
<td>proprietary</td>
<td>AGPLv3</td>
<td>GNU LGPL</td>
</tr>
</tbody>
</table>

Both Dropbox and ownCloud, provide versioning of files. Whereas ownCloud is free and open source software, Dropbox is a proprietary software application. Due to this, some of the Dropbox characteristics shown in Table 1 are assumed based on its behaviour and marked with ‘*’.

In contrast to Git, version control in Dropbox and ownCloud is file-based. They are not able to handle interdependencies of changes between two or more file. Therefore, checking out an old file version can lead to inconsistency. Furthermore, Dropbox and ownCloud do not provide any merging approaches, instead they provide a all conflicting versions of a file in parallel.
5 Conclusions

Increasing requirements and stronger integration of stakeholders raise complexity of product development process. Due to this, methods and tools that are used in product development also need to become more and more complex. As known from system theory, approaches that are based on self organisation and distributed networks are more suitable to manage complexity than central approaches. For this reason, a distributed data management approach was discussed in this paper.

It had been shown that there are some PDM approaches known which propose a distributed data management. Although the advantages of distributed PDM are known and IT technology, which is needed to realise such an application is available, distributed PDM approaches are not productively used. It was assumed that beside the extensive effort PDM provider have neither an interest nor the potential to implement such a system. For this reason, in this paper a low level approach which focuses on selected PDM functions and which bases on Git was described. It was shown that the tool is in principle suitable for use in product development. However, some specialities concerning the management of large and binary files need to be considered. While with Git LFS an extension for handling large files exists, tools that are able to merge different versions of files that are used in product development need to be developed.

Furthermore, the distributed PDM approach discussed here focuses on versioning and sharing of files. In general, PDM systems offer much more functions which are not considered in this work. Against this background, the Git-based PDM approach should not be understood as a replacement of existing PDM systems, but it can be used in a complementary manner. Future work may investigate on how this functions can be implemented without a central authority, and how the approaches can be integrated into a holistic, distributed PDM system.

Furthermore, the way of collaboration in product development needs to be revised. The distributed Git-based PDM approach describes product development as P2P network. The process participants are considered as equal. They negotiate with each other directly about the framework conditions for collaboration. This indicates a stakeholder approach for product development. In contrast to conventional approaches, which mainly focus on market or customer needs, the stakeholder approach considers the needs of all parties (both internal and external) involved in the product development process. Due to this, the stakeholder approach has significant impact on

- coordination and decision-making processes within the company,
- organisational structure of a company and
- cooperation between companies.

The issues above are not discussed in this paper.

They address many research fields including business studies and social sciences. Today companies are organised in a hierarchical manner. But as a first investigation shows, there are companies that follow a distributed, stakeholder oriented approach. For this reason, further research should also focus on analysing data management and collaboration within those companies to explore advantages as well as drawbacks.
Acknowledgements

This work is financially supported by the Linz Center of Mechatronics (LCM) in the framework of the Austrian COMET-K2 programme. Further acknowledgements are made to O. Univ.-Prof. Dipl.-Ing. Dr. Klaus Zeman (Head of Institute of Mechatronic Design and Production – IMDP) and Univ.-Prof. Dr.-Ing. Dr. h.c. Sándor Vajna (Head of Chair of Information Technologies in Mechanical Engineering – LMI) for initiation and support.

References


Distributed data management in product development using Git


