D^{4}M, a Self-Adapting Decentralized Derived Data Collection and Monitoring Framework*

Karsten Saller[1], **Dominik Stingl**[2], **and Andy Schürr**[2]

[1] karsten.saller@es.tu-darmstadt.de, [3] andy.schuerr@es.tu-darmstadt.de
Real-Time-Systems Lab, Technische Universität Darmstadt, Germany
[2] dominik.stingl@kom.tu-darmstadt.de
Multimedia Communications Lab, Technische Universität Darmstadt, Germany

Abstract: Peer-to-peer systems are evolving as a viable distributed resource sharing paradigm on the Internet. The trend is growing towards the usage of such decentralized systems because they are more scalable and resource efficient than centralized systems. Current decentralized systems, like peer-to-peer networks, lack functionality to adapt the transmission of certain information artifacts, according to their access patterns. Additionally, there is still no approach for an efficient dependency management between distributed and dependent information in decentralized networks. This paper presents our ideas of D^{4}M, a framework for the management of distributed derived data in decentralized systems, and how such data can be handled in an efficient manner.

Keywords: Decentralized Systems, Monitoring, Self-Adapting Systems

1 Introduction

Nowadays, in many application domains, the trend to use more and more decentralized IT system solutions instead of still relying on centralized client-server applications is growing. Sample applications that already rely on decentralized systems are federated data management systems in the health care sector [STMB07], globally distributed development environments, Wiki engines [MLS08], or generalized monitoring approaches [GSR$^+$09].

Recent work on peer-to-peer (P2P) overlay networks offer scalable and fault tolerant services for data management in decentralized systems [RD01]. These systems operate without any need for a centralized server and guarantee the accessibility of stored artifacts by means of sophisticated replication strategies. Unfortunately, none of the existing P2P overlays offer solutions to manage dependencies between artifacts stored on different peers in the network in order to keep derived data in a consistent state. As a consequence, a dynamically distributed derived data management (D^{4}M) facility must be developed, that extends the basic services of a P2P overlay without reintroducing any centralized dependency management servers.

When developing such a D^{4}M framework on top of a P2P data management overlay, we have to take requirements into account, like minimizing the number of recomputation steps of derived data, when a specific artifact has been changed, and disseminating the changes as early or as late as possible. Since decentralized systems, like P2P, deal with the dynamics within the network

* This work has been (partially) funded by the DFG research unit 733 QuaP2P

by self-adaptation, a distributed data management solution must also be able to adapt itself to the current environment. Until today, there is no mechanism for an efficient and self-adapting data management that can operate on graph-based communication structures like decentralized systems. It is, therefore, our goal to develop a D^{4}M framework that operates as an over-overlay on existing P2P overlays and that efficiently maintains dependencies between multiple artifacts across an arbitrary number of peers, by using self-adapting data propagation and recomputation heuristics based on locally monitored access patterns. According to these information, D^{4}M will choose the most suitable strategy to compute and propagate specific information to all dependent peers, thereby decreasing the traffic overhead.

The compiler construction community developed quite a number of algorithms in the past for the optimal recomputation of derived attributes of syntax trees and graphs in centralized systems for that purpose [SSK00]. Considering the capabilities to operate on graph structures, the extension of such incremental attribute evaluation mechanisms to decentralized systems seems reasonable. These mechanisms are characterized by operating strategies that are most efficient under particular circumstances. Hence, the presented framework will leverage incremental attribute evaluation mechanisms and adapt them to the dissemination capabilities within a distributed (and decentralized) environment.

Section 2 introduces, our research issue of distributed dependency management by using a running example. In the following Section 3 we present our framework D^{4}M and how it deals with the problems of a distributed dependency management. This section also introduces three propagation strategies in detail and gives a short comparison of those strategies. Afterwards, Section 4 presents our future work and research questions. Finally, we discuss our approach with an outline to alternative approaches and related work in Section 5, before we conclude in Section 6.

2 The Distributed Derived Data Management Problem

The D^{4}M framework can be used in various scenarios where related artifacts are stored in a distributed environment, like monitoring P2P systems [GSR$^+$09], information handling in smart grids [BD07], or distributed development environments. This section introduces the usage of D^{4}M within a distributed build environment as a running example. The example is then used to outline the main problems and the major requirements for D^{4}M.

We chose the example of a distributed build environment, since this scenario is easily understandable within the overall computer science community. However, we have to point out that a distributed build environment is only one amongst many application scenarios, we are currently examining. The presented distributed build approach represents an extension to our previous research towards a P2P-based global software development environment [MLS08].

2.1 Example

The example in Figure 1 shows one possible scenario for the management of distributed data in a P2P network. It describes an abstract distributed build environment for a simple software application, where the components are distributed and built over four different peers and have

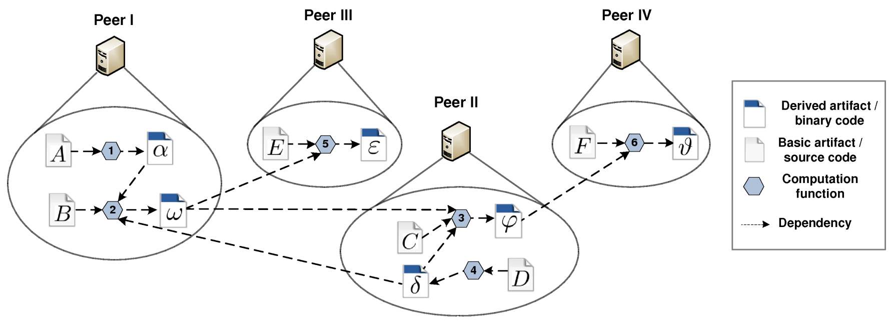


Figure 1: Distributed build example for D^{4}M within a P2P system

overlapping dependencies. This forms a dependency graph which is distributed over all participating peers. In this example, the source code of a component represents the basis of an artifact and the corresponding binary code represent the derived artifact of this basis, which is built by a computation function.

Overall there are two kinds of artifacts: Basic artifacts have no incoming dependencies, but may be used by a computation function to generate derived data, like building the source code components *A, B, C, D, E,* and *F*. Derived artifacts, or derived data, are generated by a computation function, like the binary code α, ω, φ, δ, ε, and ϑ. In the upcoming sections, the term *Basis* refers to basic artifacts, *Derivative* to derived artifacts, and *Artifact* to both kind of artifacts if a distinction is not needed.

Computation functions have only one target Derivative but they can evaluate multiple source Artifacts (Bases or Derivatives). A corresponding example is the binary code Derivative ω, which is built by *computation function* 2, and therefore requires α, *B,* and δ.

Changes in a Basis or computation function may alter the directly or transitively dependent Derivatives. Hence, changes have to be disseminated to all dependent computation functions, so that they can update their Derivatives. According to our distributed build scenario in Figure 1, a change in the source code or the computation function may alter the corresponding binary code. For example, if component *A* is changed, the *computation function 1* has to be recomputed and the Derivative has to be disseminated to the *computation function* 2. Since all needed information is stored on the same peer, the dependency is well known and ω can easily be updated by *Peer I* locally. Remote dependencies, like the one between ω and *computation function 3*, are more complicated to handle. After an update of *B*, *computation function* 2 must be recomputed and the newly derived data ω needs to be disseminated to *computation function 3* which is stored on remote *Peer II*. To keep the network in a consistent state, it has to be guaranteed, that an update is disseminated to all its dependent computation functions. Inconsistencies like two Derivatives relying on different versions of the same remote Derivative have to be avoided. In the example in Figure 1 an update of ω has to be consistently disseminated to *computation function 3* and *5*.

An *Update-Path* consists of a sequence of Derivatives in a topological order that have to be updated if one of its Basis, computation functions or required Derivatives have been changed. This path can directly be mapped to the dependency graph of the corresponding Derivative, such as in the example a Update-Path for component *D* may look like $\{D\colon \delta, \omega, \varphi, \varepsilon, \vartheta\}$.

The opposite to an Update-Path represents the *Evaluation-Path*. This path describes a sequence of Derivatives that have to provide their latest data to the target Derivative. If the Derivatives on this path are outdated, they have to request the latest derived data from Derivatives they depend on, which may be transitively nested. If in the example, the components α, ω, φ, and D are outdated and the binary code for φ is rebuild, a corresponding Evaluation-Path may look like $\{\varphi\colon \delta, \omega, \alpha\}$.

2.2 Research issues

In this section the main problems of distributed dependency management will be identified and summarized. To ensure an efficient derived data management, our approach has to deal with the following six challenges:

1. Dependency management: In order to handle distributed dependencies among Artifacts it is necessary to keep track of these dependencies locally. If a computation function or Artifact is locally changed, all directly dependent or directly required Derivatives have to be known to initiate a recomputation or at least an invalidation.

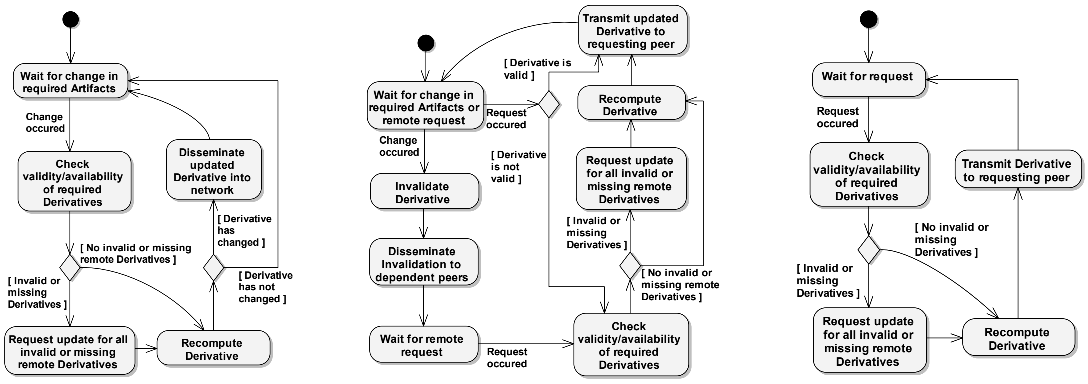
2. Guarantee of validity: After an Artifact is updated, it has to be guaranteed that the dependent Derivatives rely on the same derived data. If an Artifact is accessed, nothing may be outdated anymore. This can be done in a random and naive way, or the dissemination and necessary recomputations can be ordered according to the Update- or Evaluation-Path. This avoids unnecessary computations and network traffic.

3. No global view: Because information is stored on various peers in the network, a retrieval of information costs time. During this retrieval, the information may be outdated. It is therefore hard to collect up-to-date global information from the network, such as the distributed dependency graph for a Derivative.

4. Self optimization: To improve the efficiency of an update propagation, the most suitable algorithm has to be chosen for each Derivative according to the current situation. Since the behavior of the peers is dynamic, the frequency of accesses, updates, and references may change constantly. To realize self-optimization, local access information of the according Derivative has to be present.

5. Termination: Derivatives can rely on multiple distributed Artifacts in the network and in turn the required Derivatives can transitively depend on further Derivatives in the network. Hence, even if only one dependency is locally known, the Derivative can depend on multiple remote Derivatives. In the worst case, these dependencies form a cycle and, because global view cannot be locally provided without additional mechanisms, these cycles cannot be detected and the algorithm might end up in a non-terminating update loop.

6. Time constraints: Derivatives can rely on time constraints and therefore, have to be updated until a given deadline, like in real-time environments such as monitoring the power consumption of electrical devices (smart grids) [BD07]. Hence, during an update process, the computation of all dependent Derivatives has to be prioritized in order to accelerate the update and to meet the time constraint.


Wait for change in required Artifacts
Change occured
Check validity/availability of required Derivatives
Disseminate updated Derivative into network
[Derivative has changed]
[No invalid or missing remote Derivatives]
[Invalid or missing Derivatives]
[Derivative has not changed]
Request update for all invalid or missing remote Derivatives
Recompute Derivative
Transmit updated Derivative to requesting peer
[Derivative is valid]
Wait for change in required Artifacts or remote request
Request occured
Recompute Derivative
Change occured
[Derivative is not valid]
Invalidate Derivative
Request update for all invalid or missing remote Derivatives
[Invalid or missing Derivatives]
Disseminate Invalidation to dependent peers
[No invalid or missing remote Derivatives]
Check validity/availability of required Derivatives
Wait for remote request
Request occured
Wait for request
Request occured
Check validity/availability of required Derivatives
Transmit Derivative to requesting peer
[No invalid or missing Derivatives]
[Invalid or missing Derivatives]
Request update for all invalid or missing remote Derivatives
Recompute Derivative

(a) Activity diagram Eager (b) Activity diagram Lazy (c) Activity diagram Quiet

Figure 2: Activity diagrams handling local Derivative

3 The D^{4}M Framework

The following section describes our idea of a dynamic distributed derived data management framework which is designed to solve the described problems in the previous section. First, three Derivative-specific operating modes are presented that can be combined in different ways and that intend to deal with the distributed dependency management. Then, three different strategies are introduced and show how they can optimize the performance of the system.

3.1 Operating modes for dependency management

The most promising approach to improve the efficiency of handling distributed dependencies is the optimization of the propagation during a communication. Therefore, three different *operating modes* have been identified, in which a Derivative can be configured. The operating modes, illustrated in Figure 2, represent different possibilities to propagate information:

1. *Eager*: Possibly changed derived data is immediately recomputed and changed values are then immediately redisseminated to interested peers.
2. *Lazy*: Derived data is only (re-)computed, locally cached, and sent to other peers on request. Furthermore, messages, which invalidate dependent Derivatives (invalidation messages), are sent to peers that manage data that depend on invalidated or recomputed data of the just regarded peer.
3. *Quiet*: Derived data is recomputed for each incoming request and never cached or disseminated beforehand.

If the Derivative is in an Eager operating mode, it is always kept up-to-date. As soon as one required Basis or remote Derivative, on which the current Derivative depends, is updated, an update of the current Derivative is computed as shown in Figure 2a. To do this, all remote Derivatives, which are required to recompute this Derivative, have to be available and up-to-date.

The previous content of the updated Derivative is not immediately overwritten but cached, after the update is finished. Thereby, the old and newly computed content can be compared and only if the content has changed, the updated Derivative is pushed to its dependent peers. Not every change in a required Artifact will change the Derivative: For example, the computation functions from our distributed build in Section 2.1 could be configured to skip comments. Therefore, a change in the source code will not change the corresponding binary code.

Another possibility to propagate changes to dependent Artifacts is the Lazy operating mode which is shown in Figure 2b. In Lazy mode, the traffic is being minimized by transmitting every Artifact only if necessary, but at the same time, all depending Derivatives are informed about changes on their Evaluation-Path. If a local Basis or a remote Derivative on the Evaluation-Path is changed, the depending local Derivative is invalidated but not yet recomputed. An invalidation message is disseminated along the Update-Path of the invalid Derivative and thus, all directly and transitive dependent Derivatives are invalidated. When the invalidated Derivative is requested, it is recomputed, and therefore flagged as valid, before it is transmitted to the requesting peer. If a valid Derivative is requested, it is not recomputed but directly transmitted to the requesting peer.

The opposing approach of Eager is the Quiet operating mode, since every required Derivative has to be explicitly pulled instead of being pushed automatically in case that it has changed. In this operating mode, a Derivative will be computed for each request, which means that the actual computed Derivative is never saved, neither locally nor on the depending remote peer. If any of the directly depending Derivatives is updated, the required Derivatives, which operate in Quiet mode, always have to be explicitly pulled from the network and recomputed on the corresponding peers. The behavior illustrated within the activity diagram in Figure 2c describes the handling of a local Derivative, that is operating in a Quiet mode.

All three operating modes - Eager, Lazy, and Quiet - have their advantages and disadvantages. For example, a Lazy mechanism always has a higher latency, because it starts its working process when the resource is requested. In contrast, with an Eager approach, the work is processed immediately and the system is therefore always up-to-date, even if this is not necessary for the moment. Eventually, the Quiet mode produces no overhead, if the access frequency of the Derivative is low, since the Derivative is directly requested and transmitted, but the overhead constantly increases with a higher access frequency. Hence, Eager processing produces more overhead but provides superior time constraints, Lazy minimizes the overhead for popular Derivatives, and, Quiet is the most suitable for unpopular Derivatives.

3.2 Quantitative assessment of the traffic overhead

To provide a more detailed analysis of the three previously introduced operating modes, a comparison of the traffic consumption in Eager, Lazy, and Quiet mode for three simplified scenarios will be discussed in this section.

Comparison of Eager, Lazy, and Quiet modes To identify the best application scenarios for the three operating modes, the consumed traffic has to be compared. Because an implementation of D^{4}M is not yet available, a simplified mathematical model was used to visualize the approximated behavior. Therefore, only the additional traffic which is produced in Eager, Lazy, or Quiet mode for one Derivative and its direct remote dependencies has been analyzed. The P2P related network traffic was not considered to be of relevance in this scenario and it was assumed that all required Artifacts, for the Derivative under consideration, are locally available and always up-to-date.

The update and access patterns for a specific artifact are the most crucial parameters that influence the three operating modes (cf. Section 3.1). For example, in the distributed build scenario from Section 2.1 the update and access patterns may represent write and read operations on distributed software components. An additional parameter, that is needed to provide a meaningful comparison of the traffic behavior, is the size of the propagated data. With these assumptions the traffic may be calculated for a specific point in time t with the defined update interval I_U and access interval I_A as well as the size of the Derivative S_D and the size of the invalidation message S_{IM}. A parameter that is needed to calculate the traffic in Eager is the probability P_C that the Derivative really changed its content after an update. To include directly dependent Derivatives, the number of dependencies is denoted as N_D and the number of actual remote requests as N_{RR}, with $N_{RR} \leq N_D$. The following six formulas have been implicitly proposed to approximate the network traffic for an Eager, Lazy, and Quiet operating mode in a simplified environment.

With Proposition 1 the traffic consumption of the Quiet mode may be calculated. The Quiet operating mode strongly depends on the frequency a Derivative is being accessed and thereby relies on the number of requests. Every time it is accessed, it has to be transmitted to the requesting peers, and therefore traffic is increased by the size of the Derivative S_D times the number of requesting peers N_{RR}.

Proposition 1 *Traffic calculation for the Quiet mode*

$$Q(t) = \begin{cases} Q(t-1) + (S_D \times N_{RR}), & \mathit{if}\ t\ \mathit{mod}\ I_A = 0 \\ Q(t-1), & \mathit{otherwise} \end{cases}$$

To approximate the traffic consumption in the Eager operating mode, Proposition 2 may be used. Contrary to the Quiet mode, in Eager mode, the traffic strongly depends on the frequency I_U that the Derivative is being updated and the probability P_C that the content of the Derivative really changes after an update. Every time it is updated, it has to be disseminated to all dependent peers, and therefore the network traffic is increased by the product of the size of the Derivative S_D multiplied with the number of dependencies N_D with each changing update.

Proposition 2 *Traffic calculation for the Eager mode*

$$E(t) = \begin{cases} E(t-1) + (S_D \times P_C \times N_D), & \mathit{if}\ t\ \mathit{mod}\ I_U = 0 \\ E(t-1), & \mathit{otherwise} \end{cases}$$

The traffic approximation for the Lazy operating mode is more complicated, and therefore is divided in three propositions. First of all, it has to be calculated whether the Derivative is valid or not at time t with Proposition 3. A Derivative is either invalid, if it was valid and is now updated, or if it was already invalid and has not yet been accessed.

Proposition 3 *Validity calculation for the Lazy mode*

$$V(t) = \begin{cases} \mathit{Invalid}, & \mathit{if}\ (V(t-1) = \mathit{Valid}\ \&\ t\ \mathit{mod}\ I_U = 0) \\ & \|(V(t-1) = \mathit{Invalid}\ \&\ t\ \mathit{mod}\ I_A \neq 0) \\ \mathit{Valid}, & \mathit{otherwise} \end{cases}$$

Secondly, with information about the validity of the Derivative, the traffic consumed by invalidation messages can be calculated with Proposition 4. If the Derivative is valid and has been updated, an invalidation message is disseminated to all dependent peers, and therefore the traffic is increased by the size of the invalidation message S_{IM} times the number of dependencies N_D.

Proposition 4 *Invalidation traffic calculation for the Lazy mode*

$$VT(t) = \begin{cases} VT(t-1) + (S_{IM} \times N_D), & \textit{if } (V(i) = \textit{Valid} \,\&\, t \bmod I_U = 0) \\ VT(t-1), & \textit{otherwise} \end{cases}$$

Thirdly, the Derivative-specific traffic consumption in Lazy mode can be calculated with Proposition 5. If the Derivative is accessed in an invalid state, it has to be disseminated into the network to all requesting peers N_{RR}. Hence, the traffic is increased with each request but only if the Derivative is invalid.

Proposition 5 *Derivative traffic calculation Lazy*

$$DT(t) = \begin{cases} DT(t-1) + (S_D \times N_{RR}), & \textit{if } (V(i) = \textit{Invalid} \,\&\, t \bmod I_A = 0) \\ DT(t-1), & \textit{otherwise} \end{cases}$$

Finally, in Proposition 6 the overall traffic consumption for the Lazy operating mode at time t can be calculated by adding the Derivative and invalidation traffic at time t.

Proposition 6 *Overall traffic calculation for the Lazy mode*

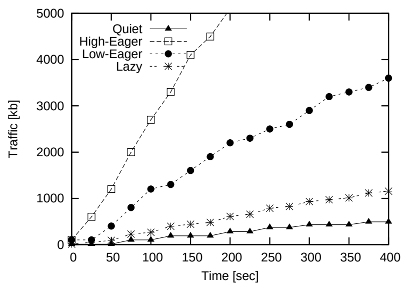
$$L(t) = VT(t) + DT(t)$$

To visualize the proposed traffic consumption over time in the described simplified scenario, the three operating modes have been compared in Figure 3. Parameters S_D and S_{IM} have been set to 10 kilobytes (kb) and 1 kb, respectively, and the scenario starts at t = 0 seconds (sec) and finishes at t = 400 sec. Handling the remote requests N_{RR}, a random value is chosen between 5 and 10 at every time t, the overall N_D is defined with 10. To show the major impact of P_C, we looked into two different Eager configurations: one assumes a high probability for a change with 70% (High-Eager) and the other one a low probability with 30% (Low-Eager).

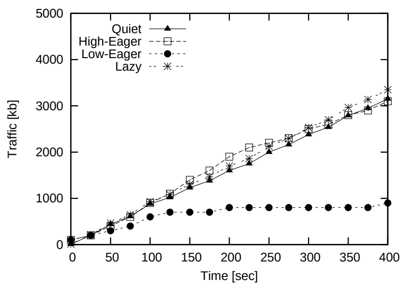
Figure 3a illustrates the first scenario, in which I_U was assumed 3 sec and I_A with 12 sec. It has been shown, that the Eager mode with a high probability for a change performed the worst, directly followed by the Low-Eager mode. Both, the Lazy and Quiet mode, performed better than the Eager modes and at the end, Quiet had the least traffic overhead under these circumstances. This can be explained by the high frequency of updates, which caused a lot of traffic in the Eager modes, because every update, that changed the content, caused a complete dissemination to all dependencies.

Figure 3b shows the traffic consumption in a well balanced scenario, where I_U and I_A were 10 sec. The three modes - High-Eager, Quiet, and Lazy - performed similar, but Low-Eager produced slightly less traffic compared to the others. This can be explained, by the small probability for a change: Only 30% of all updates really changed the content of a Derivate. The figure shows that no additional traffic was consumed between t = 200 and t = 375, because the Derivative had not changed during that time.

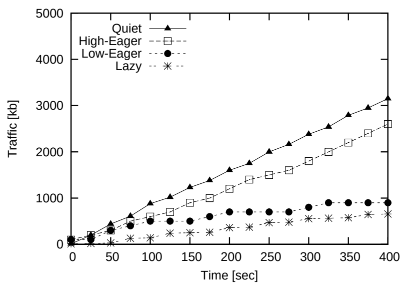
The third scenario is shown in Figure 3c, where the access interval I_A with 3 sec is four times higher than the update interval I_U with 12 sec. Under these circumstances, the Quiet mode performed worst followed by High-Eager. The Lazy mode performed slightly better than the Eager modes because it transmits the updated Derivative only to the requesting peers and the invalidation messages caused only a minimal overhead.



(a) High update frequency



(b) Equal update and access frequencies



(c) High access frequency

Figure 3: Traffic comparison Eager, Lazy, and Quiet

3.3 System Optimization

In D^{4}M the three presented operating modes will be combined in three different evaluation strategies. Each strategy will bring further improvements to the overall performance of the system:

1. *Standard algorithm:* One of the three provided operating modes will be uniformly configured in every Derivative and its computation functions. In our example in Figure 1 each binary code component is uniformly handled by the same operating mode (all Derivatives are configured to operate in Eager mode for example).

2. *Derivative-specific:* In our first step to improve the efficiency of a distributed dependency management, the operating mode for each individual Derivative may be chosen manually. If the access and update pattern for an Derivative is known, the efficiency of the system can be improved by choosing the most suitable operating mode. Otherwise, if no behavioral patterns are known, a Lazy operating mode can be chosen, since it is the only one that performs acceptably in all scenarios. To implement this strategy in our distributed build scenario, shown in Figure 1, a system architect may configure each particular binary code component individually.

3. *Profile-based:* A further improvement can be achieved, if the operating mode is chosen automatically based on meta information of the corresponding Artifacts. With this information, each Derivative will choose the optimal combination on its own, based on its update and access frequency, and the probability of a change by an update. This approach is called *profile-based Artifact evaluation*, since the local access and update behavior for each Artifact has to be monitored. A set of rules, like mapping of access and update intervals to the corresponding operating mode, has to be globally known on every peer. With these rules, the most suitable operating mode can be chosen at any time individually for every Derivative. For example, in our distributed build scenario, in every binary code component, meta data has to be gathered by monitoring-specific parameters, like build frequency or the ratio of all source code updates to those updates that modify comments only (i.e. do not require any recomputation efforts)

4 Future Work

Currently, we are implementing a first version of the D^{4}M framework and preliminary results of our research, regarding the operating modes, are encouraging. In the early stages of D^{4}M, we will evaluate the framework by simulations with PeerfactSim.KOM[5]. The results of these simulations will be used to optimize the rule sets in the profile-based system optimization strategy. After we established and configured a first operating version, we will look into scheduling and priority management mechanisms to optimize quality of service and to enable real-time communication capabilities.

Our intention is to apply the advantages of the D^{4}M framework in a monitoring mechanism for completely decentralized systems. With this approach the most recent value of various system parameters may be monitored and distributed, so that a snapshot of the current system state will be generated efficiently. According to the global system state, the system may configure itself by adapting parameters, and thus improve its overall performance. But not only the network can benefit from the monitored information. D^{4}M itself can use this information to optimize the global rule sets and operating modes, which will again influence the overall performance of the system.

Besides the given research goals, we want to answer the following research questions:

- How can globally monitored system parameters be used to our advantage?
- Do we have a benefit, if we coordinate the operating modes of dependent Derivatives?
- How can we detect and resolve cyclic dependencies?

5 Related Work and Discussion

Our techniques for disseminating dependent information, according to locally monitored access patterns, may be applied in most existing approaches that have to deal with collecting and disseminating information in decentralized systems. Therefore we provide an outline of our ideas

[5] http://peerfact.kom.e-technik.tu-darmstadt.de/

against existing approaches for self-optimization in decentralized systems in this section. Although, we presented an example for a distributed build scenario, we will concentrate on the comparison with existing monitoring approaches for decentralized systems, because this will be a major focus of our future research. Subsequently we explain why we adopted the principles of incremental attribute evaluation.

An alternative approach for managing data is represented by SkyEye.KOM [GSR+09]. SkyEye.KOM is an efficient over-overlay for information management and is applicable as a further layer on any DHT-based P2P system. The monitoring capabilities of SkyEye.KOM provide interesting possibilities for self-management features within a P2P system. The approach is specifically designed to collect statistics on the current status of the P2P system. This is done within a tree topology, where every participating peer communicates specific information about its current status, such as available disk space, to its parent peer. The parent peer aggregates all the received information with its own and forwards the aggregated information to its own parent, until the root peer is reached. In contrast to D^4M, SkyEye.KOM is more rigid and is restricted to a specific application domain, because the information is always aggregated in the same manner within the tree and only the root peer can compute the global view from the aggregated information of all participating peers.

The Push-Sum protocol represents a gossip-based monitoring and information dissemination approach [KDG03]. In gossip-based protocols, each peer randomly contacts one or more other peers in the network in each round, to exchange information. Thus, the Push-Sum protocol is a graph-based approach, like D^4M, but it can only aggregate information to create a global snapshot of the system state and not evaluate them for various purposes. In fact, the dynamics of the gossip-based information spreading leads to high fault tolerance, but the peers always communicate in the same manner and therefore lacking self-adaption.

With the D^4M framework, we have presented a promising and flexible alternative approach for monitoring and disseminating related parameters within the network effectively. Existing approaches lack flexibility and reconfigurability and operate in a rather static manner, that is the reason why a dependency management framework like D^4M is needed. We solve the issue of rigid communication by using three different operating dissemination modes - Eager, Lazy, and Quiet - and by using generalized evaluation functions instead of specific global aggregation functions. The correct usage of these three modes, according to the current system state, will save a great deal of traffic, and therefore improve the overall performance of the system.

The principle of our three operating modes originate from the field of incremental attribute evaluation, which is used during compilation of source code and to analyze programming languages [SSK00]. Incremental attribute evaluation approaches are used in todays textual programming language editors to check the static semantics and correctness of the program. The paradigm of linking dependent attributes in a graph or tree-based structure is very similar to dependency management in decentralized systems, because both, the dependencies and the distribution of artifacts, are organized in a graph structure. Hudson describes an efficient algorithm for attribute evaluation in a lazy manner [Hud91], which evaluates only those attributes that are affected by a change and that are directly or indirectly needed for the evaluation. We adopted the basic principle of this algorithm and modified it for operating in decentralized environments, because it is the only algorithm that operates on generic graph structures instead of tree-based structures.

6 Conclusion

In this paper we have presented our approach to provide distributed dependency management for decentralized systems and their possible optimization strategies. At first, an in-depth discussion about the dependency management of related information in P2P systems based on a running example has been conducted, leading to the distinction between Basis and Derivative information artifacts. Based on the research issues we identified in the presented example, we presented our D^{4}M framework which approaches the distributed dependency problem by combining three different propagation strategies and applying them according to a given rule set and locally monitored behavior patterns. These three operating modes, namely Eager, Lazy, and Quiet, induce little overhead in their target behavioral scenario, respectively. Finally we provided three different strategies for a stepwise optimization of the system until a self-sustaining and -adapting dependency management is reached. For our future work, a discussion was conducted how D^{4}M can be used to optimize existing monitoring mechanisms and, therefore, bring further improvements to existing self-optimization approaches for decentralized systems.

Bibliography

[BD07] H. Beitollahi, G. Deconinck. Peer-to-Peer Networks Applied to Power Grid. 2007.

[GSR$^+$09] K. Graffi, D. Stingl, J. Rueckert, A. Kovacevic, R. Steinmetz. Monitoring and management of structured peer-to-peer systems. *IEEE Ninth International Conference on Peer-to-Peer Computing*, pp. 311–320, Sept. 2009.

[Hud91] S. Hudson. Incremental attribute evaluation: A flexible algorithm for lazy update. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(3):315–341, 1991.

[KDG03] D. Kempe, A. Dobra, J. Gehrke. Gossip-based computation of aggregate information. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings*. Pp. 482–491. IEEE Computer. Soc, 2003.

[MLS08] P. Mukherjee, C. Leng, A. Schürr. Piki - A Peer-to-Peer based Wiki Engine. *Eighth International Conference on Peer-to-Peer Computing*, 2008.

[RD01] A. Rowstron, P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Conference on Distributed Systems Platforms*, 2001.

[SSK00] J. Saraiva, D. Swierstra, M. Kuiper. Functional incremental attribute evaluation. *Compiler Construction*, pp. 279–294, 2000.

[STMB07] N. Stolba, A. Tjoa, T. Mueck, M. Banek. Federated Data Warehouse Approach to Support the National and International Interoperability of Healthcare Information Systems. *Conference on Information Systems (ECIS)*, 2007.