Mapping Software Architecture Styles and Collaboration Patterns for Engineering Adaptive Mixed Systems

Christoph Dorn
University of California, Irvine
cdorn@uci.edu, c.dorn@infosys.tuwien.ac.at

Richard N. Taylor
University of California, Irvine
taylor@ics.uci.edu

June 2011
ISR Technical Report # UCI-ISR-11-4

Mapping Software Architecture Styles and Collaboration Patterns for Engineering Adaptive Mixed Systems

Christoph Dorn
Institute for Software Research
University of California, Irvine
cdorn@uci.edu

Richard N. Taylor
Institute for Software Research
University of California, Irvine
taylor@uci.edu

Abstract

Software architecture styles determine to large degree a system's ability to adapt during runtime. Similarly collaboration patterns describe the flexibility with which humans can join or leave a team and what constraints apply. With the rise of large-scale mixed systems, the boundary between humans and software becomes increasingly blurry. Thus, in this paper, we propose to map architectural styles to collaboration patterns. We raise awareness on how adaptation strategies in software systems might facilitate the management of Internet-scale collaborations and vice versa. Ultimately, understanding the mutual properties and interdependencies is fundamental for a holistic approach to run-time adaptation of mixed systems.

1. Introduction

Over the past twenty years we have observed a trend towards ”social software” leaving the realm of group support systems for small or medium sized teams and entering the dimension of internet-scale collaborations. We find collective intelligence and user participation among the main characteristics of the Web 2.0 [1]. Especially noteworthy is the increasingly blurry boundary between humans and software. Humans have become both provider and consumer of content and computation. Humans are no longer just the ”users” of a system but an integral part. Their interactions with other humans and software elements has a significant impact on the runtime management of software components. In mixed systems humans and software components are equal, first-class entities.

At the same time, systems have become too complex and large-scale to be managed by human administrators alone and mechanisms for self-adaptive systems have been proposed [2]. In mixed systems any adaptation mechanism now also needs to consider the human collaboration structure. As repeatedly pointed out [3], [4], architecture-based approaches to self-management address adaptation on the right level of abstraction and generality, rather than focusing on language-level or network-level adaptation. On the architectural level, adaptation actions generally describe the replacement of components and the reconfiguration of connectors. To this end, the underlying architectural style determines to a large extent the effort required to implement runtime changes [5].

We argue that the same holds true for the human collaboration structure. Collaboration patterns at design-time define what aspects change, how long adaptation takes, what the sideeffects on the collaboration status are, and what adaptation restrictions exist. Patterns at runtime enable abstraction from detailed interactions and shared artifacts to reason about the structure of the collaboration. This motivates the main contribution of this paper: mapping architectural styles to collaboration patterns. Note especially that we do not claim a perfect mapping between architectural styles and collaboration patterns, as human entities exhibit a far higher degree of autonomy and dynamicity than any pure software system.

The insights gained from this mapping are bidirectional. On the one hand, experience from building large-scale software systems can be applied to hitherto unseen internet-scale collaborations. In return, adaptation strategies in collaborative settings potentially open up new perspectives on self-adaptation in systems of massively autonomous entities.

The remainder of this paper is structured as follos. In Section 2, we discuss software runtime adaptation aspects as first introduced in [5] and how these apply in a collaborative context. These aspects then serve to characterize the mapping of architectural styles to collaboration patterns in Section 3. We apply the runtime aspects in Section 4 to analyze existing mixed systems with respect to collaboration adaptation capabilities. We discuss insights on adaptation of mixed systems and the interplay between software architectural styles

Technical Report UCI-ISR-11-4, June 2011

and collaboration patterns in Section 5 before concluding this paper with an outlook on future work in Section 6.

2. Runtime Adaptation Aspects

In our attempt to combine software architecture and human collaboration structures, it seems natural to build upon the software runtime adaptation aspects defined in the BASE framework [5], [6] — **b**ehavior, **a**synchrony, **s**tate, and **e**xecution context. We revisit the initial BASE aspects for sake of completeness, and then map the BASE aspects to the corresponding properties of collaboration patterns.

Behavior highlights the scope of supported change. This aspect concerns the means for changing a system's behavior and the respective level of abstraction (e.g., reconfiguration at code level or at component level), and whether adaptation is limited to composition of existing behaviors or new behaviors can be introduced. This also includes specification of behavior that must remain unmodified.

Asynchrony addresses the implications that come with the lag between initiating a system's adaptation and its completion. Large-scale distributed systems take potentially longer to update than compact, central systems and might never reach a completely updated status. Relevant issues also include maintenance of system constraints during adaptation and continuous system availability.

State refers to potential adaptation ”side effects” that have an impact on the system's state. Changing a data type definition might require updating all current instances to the new definition. Replacement of a component potentially involves extracting that component's state for initializing the new component with that data before the system can resume.

Execution Context raises awareness of constraints that determine whether or not adaptation can commence. For example, a component currently having control cannot be modified directly. When adaptation is time critical, however, a new, modified instance of that component called with the same input might be deployed while aborting the old version.

These aspects represent equally important concerns in the context of collaboration patterns.

Behavior similarly addresses the means for adaptation. Reconfiguration of collaboration patterns is not limited to replacing a user. Again, there are multiple levels of granularity that allow behavior adaptation. Replacement of a complete company, a team, adding another worker, or acquiring a required skill all correspond to component adaptation in software systems. Overall, adaptation goes beyond human components. Also messages between members and shared artifacts represent significant loci of adaptation. *Actions* are the equivalent to software component *Connectors* in interaction patterns. They link humans to messages, artifacts, and involved software components. Actions thereby define how to influence and drive a collaboration and simultaneously specify how flexibly the collaboration can evolve by itself.

Asynchrony in collaborations refers to the time it requires to establish a (new) team, replace a worker, become acquainted to another worker, or learn a new skill and how the joint work is affected during that phase. This aspect likewise raises awareness on constraints that need to be enforced during the adaptation. As an example requirement, there must always exist one software development team member who is familiar with any particular piece of code rather than exchanging the whole team at once.

State aspects draw attention to direct and indirect adaptation side effects when altering the means of communication, the manipulation of shared artifacts, or replacement of workers. The most knowledgable form of direct state change is loss of implicit collaboration know-how upon removing a worker. Handover of such implicit collaboration information between outgoing and incoming workers needs explicit consideration when adapting the human interaction structure.

Execution Context refers to the possibility to adapt during an active collaboration session. Whether a human may cease work on a particular task, or whether it is necessary to wait until task completion depends on multiple factors such as explicit contracts, cost and time for repeating the task, or executing compensation actions. The same holds true for the degree of coupling between two or more workers during an ongoing interaction.

3. Style-Pattern Mapping

For each mapping of architectural style to interaction pattern, we provide a short description of the architectural style followed by a longer discussion of the pattern. We also supply real world example systems that support the respective pattern.

3.1. SOA/(C)REST vs Secretary

The basic components of the World Wide Web are clients requesting data and services from a set of servers. The REpresentational State Transfer (REST) style defines: URLs as the sole means of identifying resources, a limited set of resource manipulation primitives, state-less communication interactions, and meta data describing the exchanged resources [7]. Service-oriented Architectures (SOA) follow the same client/server paradigm and share some of the REST principles but focus on providing operations (i.e., services) rather than content [8]. Computational REST (CREST) aims to go beyond a mere combination of the simplicity of REST and the computational aspects of SOA [9]. CREST enables seamless interleaving of content and computation thereby supporting the (partial) execution of services not only on the server side but also on the client side. All three styles encourage the use of intermediaries such as proxies [10] or brokers [11] for adaptation purposes.

In human collaboration, secretaries (or assistants) take on a role similar to SOA/(C)REST intermediaries. Clients cannot contact the desired person directly. Instead, they pass their request to the secretary first, who forwards the message to the principal and relays the response back to the client. Alternatively, secretaries may respond immediately on behalf of the principal. Principals in turn act as clients of their superiors. Depending on the particular application, secretaries serve as load balancing proxies, protection proxies, caching proxies, or brokers [12].

Behavior: Clients are volatile and expected to contact a secretary unannounced.Secretaries and principals may be replaced anytime, however their relation is longer lasting than between client and secretary. The set of supported message types and artifact types varies for each secretary and principal instance. Associated Secretaries and principals, however, necessarily support overlapping type sets. A single secretary may be coupled to more than one principal and vice versa.

Asynchrony: The secretary may decide on how to respond and on how to route the request depending on the client's request and the principal's availability. In case neither principal nor secretary are able to respond, the secretary can ask the client to repeat the request at a later time. Where supported, the secretary may offer to put the request on hold and notify the client once the principal becomes available again.

State: Secretaries are initially stateless and build up any required internal state from interactions or querying the principals for preferences. Having the new secretary temporarily observe the current secretary enables rapid establishing of state. Alternatively, a secretary can refer clients to another secretary during replacement.

Execution Context: A principal may decide to refuse responding to any requests, forcing the secretary to reply on behalf or reroute the request to another principal. A secretary needs to complete all current requests before replacement but can redirect new requests to other assistants.

3.2. Publish-Subscribe vs Mailinglists

In the pub/sub style, publishers and subscribers communicate indirectly by means of events and are typically unaware of each other's identity [13]. Traditionally, a message oriented middleware (e.g., a message bus) manages event collection and distribution. Subscribers process the received events and may produce new events of their own. Such strong component decoupling enables simple adaptation through runtime adding and removing of event producers and consumers [14]. Additional event filtering allows for more fine-grained adaptation [15].

In collaborative environments, the main purpose of mailing lists (or newsletters) is pushing information of general interest to a larger audience. Mostly in the form of emails (e.g., Listserv [16]), this collaboration pattern comes in different flavors, characterized by the anonymity of sender and/or receivers, the ability of receivers to reply or post their own message, and whether the list is topic or person-centric. In recent years, micro-blogging platforms such as Twitter have become a popular tool for rapidly disseminating information in large-scale environments [17]. In contrast to purely topic-centric subscriptions in pub/sub, subscriptions in micro-blogging environments are mostly person-centric. Here additional filtering capabilities are called for to distinguish between truly relevant events and noise.

Behavior: With topic-centric lists, publishers and readers may dynamically join and leave while the respective list remains unaffected. The lifetime of a person-centric list usually remains coupled to the publishing activity of its respective author. Simultaneously, new topics emerge and existing topics loose their relevance.

Asynchrony: Changes to a list's subscriber base have no side-effects. Single changes are instantaneous and require no synchronization with other users. Author removal from person-centric lists and single-publisher lists may cause receivers to resort to alternative event sources. However, support mechanisms for subscribing to relevant lists is outside the pattern's scope.

State: The collaboration domain and purpose defines the requirements for state transfer upon swapping publishers. Consumers build their internal state as new messages arrive. Most mailing list and micro-blogging implementations maintain a history of past messages that enable immediate state reconstruction.

Execution Context: Message dispatching and receiving is considered atomic. Hence, independent unidirectional messages pose no restrictions on the replacement of publishers or readers. When multiple individual messages from multiple authors create a discussion thread the removal of an involved publisher requires other authors to compensate. Newly joining readers need to build the discussion thread from historical records or wait for the end of the discussion.

3.3. Tuple-spaces vs Collaborative Editing

Tuple-spaces are similar to the publish-subscribe style with respect to strong decoupling of producers and consumers. Instead of messages, a tuple-space manages storage and retrieval of data items (i.e., artifacts) thereby imitating a (distributed) shared memory [18]. Consumers have potential write access to manipulate the shared artifacts. Any update becomes visible to other consumers including the initial producer [19].

Collaborative editing describes any type of activity where participants communicate predominantly through the manipulation of a shared artifact. Research in the domain of CSCW focused early on such capabilities in the context of shared workspace systems and groupware systems [20]. Several approaches were proposed for large-scale environments [21], until internet-scale collaborative editing became a universal success with the emergence of wikis [22]; the most prominent example being Wikipedia.

Behavior: Authors and collaborators may join or leave the workspace at any time. Collaborators are free to create new shared artifacts (thereby becoming authors) or manipulate existing ones. Typically, only authors can remove the artifacts they own. Altering the shared artifact's type is usually limited to new artifact instances. How to obtain artifact access rights is outside the pattern's scope.

Asynchrony: Where upgrading an existing artifact is envisioned, authors and collaborators need to wait while the upgrade takes place. A permanently leaving author needs to hand over ownership for each of his artifacts to at least one collaborator. Replacement of regular collaborators requires termination of all their artifact manipulation activities.

State: The shared artifact maintains the collaboration status. The collaborative editing system needs to disclose the manipulation log when collaborators require the artifact's history to construct their internal state. Alternatively, swapping two collaborations requires an out-of-band mechanism to exchange their internal states.

Execution Context: Multiple concurrent write requests need queuing and/or duplication of the item, subsequent parallel manipulation, and finally merging. The collaborative editing pattern lacks any primitives to coordinate mutually depending artifact read and write actions by distributed collaborators.

Online discussions forums [23] ,bulletin boards, and blogs [24] populate the spectrum between Mailinglists and Collaborative editing. Knowledge is distributed from producers to consumers, who in turn raise the message to the level of a shared artifact through refinement or extension in the form of commentary. Discussion topics, and blog entries are thus more than a simple broadcasted messages but at the same time do not offer the full range of manipulation capabilities that come with shared artifacts.

3.4. Components and Connectors (C2) vs Organizational Control

C2 is an event-based style to decouple components via explicit connectors [25]. Components cannot interact directly but hand over any events and requests to connectors which pass them on to the respective destination component. Components form a hierarchy where lower-level components request operations from higher-level components. Communication down the hierarchy, however, is limited to notifications.

The C2 style describes also inverse hierarchical organizations, with managers situated towards the bottom and workers towards the top. The two interface types in C2 — upward requests and downward notifications — corresponds to two functions of organizational control [26]: behavior control and output control. The supervisor applies (pro-active) behavior control (i.e., upward requests) to trigger a specific, desired behavior. Output control describes the manager monitoring work progress (i.e., downward events) to reactively maintain, or respectively restore, work performance. Requests and notifications are not addressed directly to individual employees but roles. Hence, roles are bound to particular capabilities thereby decoupling the request addressing from the actual person executing the request.

Behavior: Organizational structures lack the same degree of loose coupling that C2 Connectors achieve. Assistants can be compared to connectors but are usually not universally implemented (i.e., typically only in management). Organizational charts describe the links between the different roles and the mapping to actual humans but cannot provide request and event buffering. Adaptation corresponds to structural updates of the organization chart in terms of reassigning employees to roles and rewiring role relations.

Asynchrony: When replacing an employee or rewiring a role, higher-level subordinates need to buffer notifications until adaptation has completed. Likewise, lower-level supervisors will not receive notifications for the adaptation duration and have to refrain from dispatching requests to the affected employee. With assistants in place, responsibility for buffering of requests and notifications is transferred from employee to assistants.

State: Individual employees maintain their own state. State transfer is only required between two switching employees when collaboration know-how remains employee internal and cannot otherwise be made explicit. New employees can apply the organizational chart to contact colleagues to retrieve (additional) information necessary for constructing state. Organizations typically store general purpose state information in corporate ”data warehouses” at the highest relevant level in the hierarchy. Accounting data, for example, remains low at the management level, whereas travel guidelines are placed at the top of the hierarchy available to every employee.

Execution Context: Any ongoing request needs to be completed before an employee can be replaced. If an employee becomes unavailable during request execution, the request has to be addressed again with her substitute employee.

3.5. Map-Reduce vs Master-Worker

Map-Reduce leverages parallel computing by dividing a task into multiple independent subtasks [27]. A central coordinator node subsequently assigns each subtask to a different processing machine (map phase). In the reduce phase, the coordinator collects the individual task outcomes and distributes them again for aggregation to generate the final result.

Partitioning tasks also works well in human collaboration when the resulting subtasks remain independent. The Master defines the individual work packages. In contrast to Map-Reduce, task distribution occurs both in push and pull style. The former procedure has tasks directly assigned to Workers, whereas the latter procedure enables workers to choose which task they prefer to work on. The large-scale deployment of the Master/Worker pattern is often referred to as *crowd sourcing* [28], the most prominent example being Amazon Mechanical Turk.

Behavior: The master decides upon the number of workers that can work in parallel on the same task artifact. In the pull-style assignment, workers choose which tasks to perform, and whether to return a task unfinished. In push-style assignment, workers receive new tasks in their inbox but may still have the option to reject or delegate a task.

Asynchrony: A task artifact completely decouples master and workers. The master has the option to reassign the task to another worker (or make it available again) when the worker fails to complete the task in a predefined time-frame. The master has the responsibility to schedule multiple identical tasks for sake of reliability, or issue multiple sequential tasks until the desired result quality has been achieved.

State: The task artifact contains the complete collaboration state. Replacement of workers has no side-effect on the state.

Execution Context: All workers execute their task independently, hence, no synchronization of results is required. Multiple workers assigned to the same task artifact work on distinct copies and have no knowledge about each other. They remain similarly unaware of any replacement of the master. The new worker simply obtains the task description and commences task execution independently of any previous work done.

3.6. Peer-to-Peer (P2P) vs Self-organization

The P2P style assumes a large number of entities, each offering the same service they simultaneous request from their peers [29], but not possessing necessarily the same state. The major motivation behind P2P is resilience to node failures. To this end, P2P systems usually lack a centralized control mechanisms and expect peers to arbitrarily join and leave the network. Instead, nodes maintain a limited list of neighboring peers for exchanging and updating their view of the network. In sufficiently large networks, super peers have proven to increase efficiency by providing higher-level coordination services (potentially forming an overlay peer-to-peer network themselves). The dominant application area of P2P technologies is content distribution networks [30].

The P2P style appropriately describes self-organizing teams where no dedicated coordination structure exists. Instead, individual members collaborate in an ad-hoc fashion to complete a common goal. The flexibility provided by P2P allows individual members to become more active whenever they see fit without incurring much coordination overhead. Newcomers receive briefings from multiple members and thus are quickly brought up-to-date. Multiple communication paths maintain the information flow in case a member becomes temporarily or indefinitely unavailable. Many open-source software development efforts are structured in a P2P fashion which allows them to form self-organizing teams [31]. Self-organization, however, also has its downside as the effect of (flash)mobs demonstrates. Another challenge poses the mobilization of sufficiently many participants to keep the collaborative momentum alive.

Behavior: In a self-organizing team, every member is free to leave any time, and new members join spontaneously. Existing members are able to form arbitrary links to any other member, e.g., send messages to any other member they consider a relevant receiver. Stable members are suitable candidates for super peers but not expected to serve that role permanently. In addition usually super peers introduce new message types to improve collaboration efficiency.

Asynchrony: Typically only a (small) subset of members fluctuates at a single point in time which hardly disrupts the team. Retirement of super peers potentially degrades a team's performance until another member assumes the vacant position. The uptake of new message types occurs gradually as the update percolates through the network.

State: Collaboration state is generally spread about multiple members (i.e., those that were involved in joint activities). However, additional mechanisms for externalizing member state such as shared artifacts or message history is outside the scope of the pattern. New members gather the required state information from existing peers.

Execution Context: Members engage in joint activities and are free to leave at any time. They are, however, expected to either complete the task before leaving or at least externalize the task relevant part of their internal state. Abruptly leaving members may result in lost collaboration know-how.

4. Example Systems

In the following subsections, we present some real world examples of mixed systems and analyze how the underlying collaboration patterns influence the support for adaptation.

4.1. Amazon Mechanical Turk

Amazon Mechanical Turk [1] (MTurk) is one of the most prominent examples for crowd sourcing. The MTurk platform implements the Master-Worker collaboration pattern, enabling *Requestors* to register a *Human Intelligence Task* (HIT). *Workers* are subsequently able to search through the set of registered HITs and *accept* HITs they are interested in. Besides task specific details, a requestor configures the HIT's lifetime, monetary reward, duration, the maximum number of assigned workers, and required worker skills. Once the worker has accepted a HIT, he has the allotted time (duration) to complete his copy of the task (i.e., his *assignment*). Alternatively the worker can *return* the HIT within the allotted time, or let it time out. The HIT is then considered *abandoned*. Returning and abandoning reflects in a worker's reputation. The requester retrieves the submitted assignment and *approves* it when the result quality is satisfactory, or otherwise *rejects* it without paying the worker.

1. `http://www.mturk.com`

Runtime adaptation focuses on those mechanisms that provide for a satisfying collaboration of all participants; both requestors and workers. The requestor aims for timely and high quality HIT execution. The worker's primary goal is finding interesting and suitable HITs and receiving pay for the work performed.

Behavior: As outlined in Section 3.5, the Master-Worker pattern's main form of adaptation lies in the replacement of workers, the rewiring of workers to tasks, and the manipulation of the task artifacts. As MTurk supports only pull-based task assignment and features a dynamic workforce that cannot be directly controlled, all adaptation actions unfold their effects indirectly through HIT manipulation.

Requestors have full control over the creation, reconfiguration, and removal of HITs. They regulate the wiring of workers to HITs through (i) direct blocking of workers and (ii) specification of required skills. The requestor has full control over the HIT relevant skills and can assign and revoke the corresponding worker qualification anytime.

Workers have less control over HITs per se, but typically filter HITs for matching qualifications, rewards, and duration. In MTurk, only workers can establish a collaboration through accepting an assignment. Although they are able to return or abandon a HIT, this reflects negatively on their reputation.

Asynchrony: The crowd sourcing environment of MTurk does not foresee explicit replacement of a unresponsive or low quality worker. In the former case, MTurk automatically makes the HIT available again once the work duration has expired. In the second case, the requestor needs to register the HIT again. The requester decides on appropriate HIT values for the number of simultaneous workers, reward, and duration to promote timely and high-quality results. Unattractive rewards or unrealistic timelines coupled with a lack of backup workers leads to delays in HIT execution.

Similarly, MTurk addresses unresponsive requestors.

Each HIT specifies a deadline after which an assignment is automatically approved.

State: Workers have read-only access to the HIT description. They also remain unaware of any other worker having (currently or previously) accepted the respective HIT. Returning or abandoning a HIT has thus no side-effect on any other worker assigned in parallel. The assignment results remain with the worker until he explicitly submits his work. No modifications are possible thereafter. The requester obtains read access to an assignment as soon as it is submitted. Approval and rejection are independent of other assignments of the same HIT but multiple results are usually compared for quality assurance purposes.

Execution Context: The collaboration between requestor and worker ends upon assignment approval or rejection. Both parties can prematurely terminate a HIT anytime. As outlined above, a worker can return or abandon the HIT with reputation side-effects. The requestor has the options to *force expire* the HIT such that it becomes invisible to new workers but allows existing assignments to be completed. The result is then open for approval or rejection. Alternatively the requester can *disable* a HIT which removes the HIT and automatically approves all submitted and pending assignments.

4.2. Twitter

Twitter [2] is a micro-blogging platform used to create and maintain a social network [17] as well as distributing news [32]. Twitter supports collaboration according to the Publish-Subscribe style: every user is simultaneously a publisher and subscriber. A user subscribes to another user's twitter feed, thereby becoming a *follower* of the other user.

When a user posts a message — denoted a *tweet* which may be up to 140 characters — it becomes visible on her profile page (if the account is public). At the same time, the tweet is pushed to the *home timeline* of all her *followers*. Twitter supports message meta information in the form of *hash tags* as topic descriptors (e.g., #music), references to other users (so-called *mentions*, e.g., @BarackObama), and URL shorteners to link to external resources. Users apply *retweeting* to refer to a previous messages, or *replies* to respond to another user.

Behavior: Run-time behavior adaptation focuses on managing subscriptions — i.e., manipulation of the follower list artifact — and on message content. Subscribers are free to follow any publisher with a public profile. Subscription requests for private profiles need the publisher's approval. Twitter provides to each publisher a list of followers. The publisher has the option to block a particular subscriber. This prevents messages from showing up on the subscriber's home timeline, but they remain available to everyone on the publisher's public profile.

Although each publisher is associated with a single account and thus profile, she typically distinguishes different message types using hash tags. New hash tags may be introduced by any publisher anytime. Users then search for a particular hashtag and subscribe to the respective tweeter feeds to become followers.

Asynchrony: An unavailable publisher has no direct effect on her subscribers. Followers merely cease to receive events from that publisher but continue to receive events from other publishers. A subscriber needs to search for an alternative message source in case a publisher is his single sources of (a particular type of) continuously required events. Offline followers receive the latest events upon reconnecting to Twitter. They are not guaranteed to receive all messages after an extended duration of absence as Twitter does not provide the complete history but only the recent messages. The subscription operation is typically instantaneous and new messages are accessible immediately. Messages are delayed until approval when requesting to follow a user with private profile.

State: Users gather information (i.e., state) from public profiles available before following the publisher. Hence, a user having just joined Twitter can easily perceive which existing users to follow. On the publisher's side, messages remain available even when there are no followers.

Execution Context: There are no constraints for (un)following a publisher or blocking a subscriber. Merely private profiles require approval before a relation is established.

While the main collaboration pattern is publish/subscribe, twitter also supports *direct messages* that remain private between the sender and receiver. A private message can only be sent to followers, has the same 140 character limit as a tweet, and can be deleted from the receiver's inbox by the sender.

5. Discussion

Differences within the patterns become manifest in current collaboration frameworks. Support for large-scale collaborative work remains unsatisfactory whereas examples of large-scale deployments exist for all architecture styles. This hints at exploitable synergies between styles and patterns.

2. `http://twitter.com`

5.1. Differences between Patterns

The presented collaboration patterns address different approaches for coordinating work, communication between users, and disseminating information.

They differ in which users can be replaced or rewired and who has control over the respective (adaptation) actions. The *Master-Worker* pattern enables Masters to decide over which worker may work on a particular task. In pure *Self-organizing* teams, no worker has explicit control over any other collaborator.

Changes may have little to no effect on the overall collaboration or potentially bring it to a complete standstill. A new *Secretary* might break access to a whole range of principals, while a leaving worker might cause only a small delay in the *Master-Worker* pattern. In *Collaborative Editing*, an unavailable worker has typically no impact.

Some patterns support explicit externalization of state while others neglect this aspect. The *Master-Worker* pattern defines explicit task artifacts. *Mailinglists* frequently keep a history of recent messages while *Organizational Control* has no explicit notion of collaboration state.

Finally, some patterns allow users to freely leave and join the collaboration at any time, while certain patterns come with explicit, non-trivial side-effects. Replacing a *Secretary* or Principal requires more preparation than merely adding a new Worker to a Master. A *Mailinglist* consumer can leave any time without effecting other subscribers, whereas a member of a *Self-organizing* team commonly needs to externalize his collaboration state first.

5.2. Scalability of Styles and Patterns

In recent years we have found applications and frameworks for each of the discussed architectural styles that support deployment in a large-scale fashion. *REST* represents the fundamental style of the World Wide Web. *Publish-Subscribe* systems such as Siena bring together thousands of publishers and event consumers. *Tuple-spaces* serve as the underlying coordination mechanism in massive agent-based systems [33]. The Myx framework[3] supports the implementation of *C2* style applications. Hadoop is one implementation of the *Map-Reduce* style supporting petabyte sized data warehouses [34]. Finally, Bittorrent is an exemplary protocol for efficient internet-scale *P2P* content distribution [35].

3. http://www.isr.uci.edu/projects/archstudio/myx.html

General purpose technical systems for large-scale human collaboration do not cover the collaboration patterns to the same extent. On the simple side, message-centric systems such as Twitter and Google Groups provide pub/sub facilities. Email, Skype, and XMPP are examples for human P2P communication. Social networks and blogging sites blur the boundary between messages and artifacts. Wikis and source code repositories support *Collaborative editing*. Crowd sourcing applications such as MTurk realize the *Master-Worker* pattern but severely limit human interactions. Other task-centric frameworks such as Bugzilla remain very domain specific without becoming true *Collaborative editing* environments. At the internet-scale level, we find no systems or frameworks that support the *Secretary* pattern, *Organizational control*, or *Self-organizing* teams that go beyond message-based interaction. Note that these patterns are nevertheless supported by groupware systems targeting small and medium-scale environments. Furthermore, P2P style collaborations in software developer teams exhibit emergent behavior such as secretaries (SOA/(C)REST) or managers (C2). However, corresponding tool support in large-scale settings is virtually non existent.

As we explore new directions to cover yet unsupported patterns in large-scale settings, we should apply lessons learned from designing and building large-scale software systems. As we point out in the next subsection, insights apply in both directions.

5.3. Synergies between Styles and Patterns

As the authors of the original BASE framework have pointed out [6]: the vital mechanisms for architecture-driven run-time adaptation are (i) identification of the exchangeable parts and rendering them malleable, (ii) managing the interactions involving those parts, and (iii) explicit state management. We observe the same strategies in human collaborations: (i) establishing the involved roles and their respective adaptation authority, (ii) capturing and externalizing collaboration know-how, and (iii) promoting explicit shared artifacts (especially task descriptions).

Adaptation plans in pure software systems are typically simply enacted. This approach, however, cannot be directly applied to human collaborations. The rules and implications of automatic collaboration adaptation actions need to be known upfront to all involved parties. Most often, however, individual users retain a significant amount of control over their environment. Here, we can only give recommendations on what to do, e.g., who to contact, how to filter information, or what artifacts to manipulate. Adaptation mechanism addressing such human collaborations are thus becoming candidates for designing new techniques for managing software system comprising massive, autonomous, unreliable components. Individual, self-governing components cannot be directly manipulated but accept only recommendations. They might follow the recommendations or choose to ignore them based on internal, unobservable constraints. Subsequently, human-inspired properties such as trust, reputation, or cooperativity potentially become applicable to software components. Software entities might take context into account when deciding upon cooperation. In turn, uncooperative components may face resource restrictions or have access limited to noncritical resources.

When we choose a collaboration pattern during system design, we have to keep in mind that a single pattern usually fits only very simple, independent tasks. Complex work — be it large-scale, under-specified, or distributed — calls for more freedom and flexibility and thus requires combination of multiple styles. Take platforms for open source development such as GitHub or Sourceforge as an example: they provide an integration of wikis, mailing lists, code repository, and bug tracking. Note in general that the presented patterns provide idealistic structures for understanding the implications of system design choices. Most real world applications integrate two or more collaboration patterns to cover complementary needs. Facebook, for example, primarily adopts a publish-subscribe/mailinglist style (3.2) for personal status updates and enables additionally a P2P style (3.6) text chat between friends for more focused communication.

Thus when collaboration complexity increases, externalizing state becomes ever more important as we have to trade-off strong decoupling for the benefits of joint creativity. This immediately raises privacy, respectively security issues. Large-scale collaborative environments in particular need to take up design principles from software systems — e.g., least privilege, complete mediation, separation of privilege — and apply them to human entities. In return, the mechanisms how humans establish trust through joint, repeated interactions provides an alternative, self-regulating security approach in open software systems.

6. Conclusion

Architectural styles and collaboration patterns share the same run-time adaptation aspects: behavior, asynchrony, state, and execution context. We applied these aspects to highlight how architectural styles can be mapped to collaboration patterns and vice versa. We hope that becoming aware of the similarities helps to cross-pollinate adaptation strategies for software systems and collaborative work. Fault-resilient architectural styles such as P2P or tuple spaces provide us techniques for understanding and governing self-organizing open source teams or large-scale collaborative editing. In return, collaboration patterns that leverage human dynamics and autonomy provide concepts to address adaptation of systems comprising multiple, independent software components.

Two complementary future efforts build upon this mapping of styles and patterns. On the one hand, we intend to refine the aspects that currently describe large-scale, dynamic systems (e.g., WWW, protein networks) to better characterize internet-scale collaborations. We expect relevant aspects to include the power-law link distribution, the rich-club phenomenon, and network motifs. On the other hand, we have begun work on an architectural description language that integrates the human collaboration structure and software architecture. Ultimately, such a description will enable reasoning about the effects of software reconfiguration on human collaboration and vice versa, thereby realizing a true, unified adaptation of the overall mixed system.

Acknowledgment

This work is supported in part by the National Science Foundation under grants CCF-0917129, CCF-0820222, and CCF-0808783 and the Austrian Science Fund (FWF) under grant number J3068-N23.

References

[1] T. O'Reilly, "What is web 2.0: Design patterns and business models for the next generation of software," MPRA Paper 4578, Mar. 2007.

[2] M. C. Huebscher and J. A. McCann, "A survey of autonomic computingdegrees, models, and applications," *ACM Comput. Surv.*, vol. 40, pp. 7:1–7:28, August 2008.

[3] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, pp. 54–62, May 1999.

[4] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *International Conference on Software Engineering*, 2007, pp. 259–268.

[5] R. N. Taylor, N. Medvidovic, and P. Oreizy, "Architectural styles for runtime software adaptation," in *WICSA/ECSA*, 2009, pp. 171–180.

[6] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Runtime software adaptation: framework, approaches, and styles," in *Companion of the 30th international conference on Software engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 899–910.

[7] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, pp. 115–150, May 2002.

[8] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, pp. 38–45, November 2007.

[9] J. R. Erenkrantz, M. M. Gorlick, G. Suryanarayana, and R. N. Taylor, "From representations to computations: the evolution of web architectures," in *ESEC/SIGSOFT FSE*, 2007, pp. 255–264.

[10] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[11] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing*. Chichester, UK: Wiley, 2007.

[12] S. Dustdar and T. Hoffmann, "Interaction pattern detection in process oriented information systems," *Data Knowl. Eng.*, vol. 62, pp. 138–155, July 2007.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[14] L. Fiege, F. C. Gärtner, O. Kasten, and A. Zeidler, "Supporting mobility in content-based publish/subscribe middleware," in *Middleware*, 2003, pp. 103–122.

[15] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '01. New York, NY, USA: ACM, 2001, pp. 115–126.

[16] D. Grier and M. Campbell, "A social history of bitnet and listserv, 1985-1991," *Annals of the History of Computing, IEEE*, vol. 22, no. 2, pp. 32 –41, apr-jun 2000.

[17] D. Zhao and M. B. Rosson, "How and why people twitter: the role that micro-blogging plays in informal communication at work," in *Proceedings of the ACM 2009 international conference on Supporting group work*, ser. GROUP '09. New York, NY, USA: ACM, 2009, pp. 243–252.

[18] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, January 1985.

[19] G. P. Picco, D. Balzarotti, and P. Costa, "Lights: a lightweight, customizable tuple space supporting context-aware applications," in *Proceedings of the 2005 ACM symposium on Applied computing*, ser. SAC '05. New York, NY, USA: ACM, 2005, pp. 413–419.

[20] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '89. New York, NY, USA: ACM, 1989, pp. 399–407.

[21] F. Pacull, A. Sandoz, and A. Schiper, "Duplex: a distributed collaborative editing environment in large scale," in *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, ser. CSCW '94. New York, NY, USA: ACM, 1994, pp. 165–173.

[22] A. Kittur and R. E. Kraut, "Harnessing the wisdom of crowds in wikipedia: quality through coordination," in *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, ser. CSCW '08. New York, NY, USA: ACM, 2008, pp. 37–46.

[23] L. A. Adamic, J. Zhang, E. Bakshy, and M. S. Ackerman, "Knowledge sharing and yahoo answers: everyone knows something," in *Proceeding of the 17th international conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 665–674.

[24] B. A. Nardi, D. J. Schiano, M. Gumbrecht, and L. Swartz, "Why we blog," *Commun. ACM*, vol. 47, pp. 41–46, December 2004.

[25] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins, "A component- and message-based architectural style for gui software," in *Proceedings of the 17th international conference on Software engineering*, ser. ICSE '95. New York, NY, USA: ACM, 1995, pp. 295–304.

[26] W. G. Ouchi and M. A. Maguire, "Organizational control: Two functions," *Administrative Science Quarterly*, vol. 20, no. 4, pp. pp. 559–569, 1975.

[27] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, January 2008.

[28] D. C. Brabham, "Crowdsourcing as a model for problem solving," *Convergence: The International Journal of Research into New Media Technologies*, vol. 14, no. 1, pp. 75–90, 2008.

[29] A. Oram, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001.

[30] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, pp. 335–371, December 2004.

[31] K. Crowston, Q. Li, K. Wei, U. Y. Eseryel, and J. Howison, "Self-organization of teams for free/libre open source software development," *Inf. Softw. Technol.*, vol. 49, pp. 564–575, June 2007.

[32] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[33] G. Cabri, L. Leonardi, and F. Zambonelli, "Mars: a programmable coordination architecture for mobile agents," *Internet Computing, IEEE*, vol. 4, no. 4, pp. 26 –35, jul/aug 2000.

[34] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using hadoop," *Data Engineering, International Conference on*, vol. 0, pp. 996–1005, 2010.

[35] D. Qiu and R. Srikant, "Modeling and performance analysis of bittorrent-like peer-to-peer networks," in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 367–378.