Editor: Schahram Dustdar • dustdar@infosys.tuwien.ac.at

The Social Compute Unit

Schahram Dustdar • *Technical University of Vienna*
Kamal Bhattacharya • *IBM Research — India*

Social computing is perceived mainly as a vehicle for establishing and maintaining private relationships and thus lacks mainstream adoption in enterprises. Collaborative computing, however, is firmly established, but no tight integration of the two approaches exists. Here, the authors look at how to integrate people, in the form of human-based computing, and software services into one composite system.

Business process management (BPM) and workflow systems have had tremendous success in the past two decades with respect to both mindshare and deployment. We can safely consider service-oriented architecture (SOA) — BPM's most recent manifestation — to be a "business-as-usual" design practice. On the other hand, we're observing enterprises exploring, if not even embracing, social computing as an alternative for executing more unstructured yet team-based collaborative, outcome-based strategies. Gartner predicts that by 2015, we'll observe a deeper penetration of "social computing for the business" as more enterprises struggle to deal with the rigidity of business process techniques (www.gartner.com/it/page.jsp?id=1470115). Such methods are suitable for menial tasks but inflexible when it comes to supporting business users who must deal with more complex decision making. However, a huge gap clearly exists between BPM's technologies, usage patterns, and workflows on the one hand, and social computing as it's known today.

Toward Social Work Styles

Workflow technologies have the ability to monitor and measure the execution of well-defined work units that lead to a well-defined, repeatable outcome. In a way, workflow technologies are similar to programs, and humans are an essential element of the instructions used in those programs. However, workflow technologies have difficulty supporting more complex business-decision work styles and novel dynamic interaction patterns. In such patterns, the process is hard, if not impossible, to define, and it might include emerging teams of socially networked groups not known at design time.

On the other hand, current social computing methods and technologies (instantaneous information exchange through social networking platforms, microblogging, and so on) work on the instruction level (of programs) and are by design suitable for generating more complex outcomes due to their inherent flexibility. However, businesses have yet to determine how to integrate such technologies into larger programs. These Web-scale systems would require support from the whole spectrum, from ad hoc collaborations of nimble teams to support for structured interactions and work styles suited for today's global business realities.

These challenges are associated with the perceived statistical variability of generating a well-defined outcome juxtaposed with the deterministic outcomes of business process workflows. Hence, we argue here that today's enterprises are hesitant to bring current social computing techniques into the mainstream of their organizations. Our goal is to advance current social computing techniques and approaches

 Published by the IEEE Computer Society 1089-7801/11/$26.00 © 2011 IEEE

by proposing a novel concept: the *social compute unit* (SCU).

Our approach includes both human- and software-based computing in one coherent conceptual framework that allows for programming and instantiating composites of human-provided services (HPSs) and software-based services (for example, Web services).[1,2] Human-based computing has tremendous potential, yet we must be more descriptive about how to program a human-based system. At the same time, we postulate that software will never be perfect, so it's imperative to understand the behavior of a combination of human- and software-based computing.

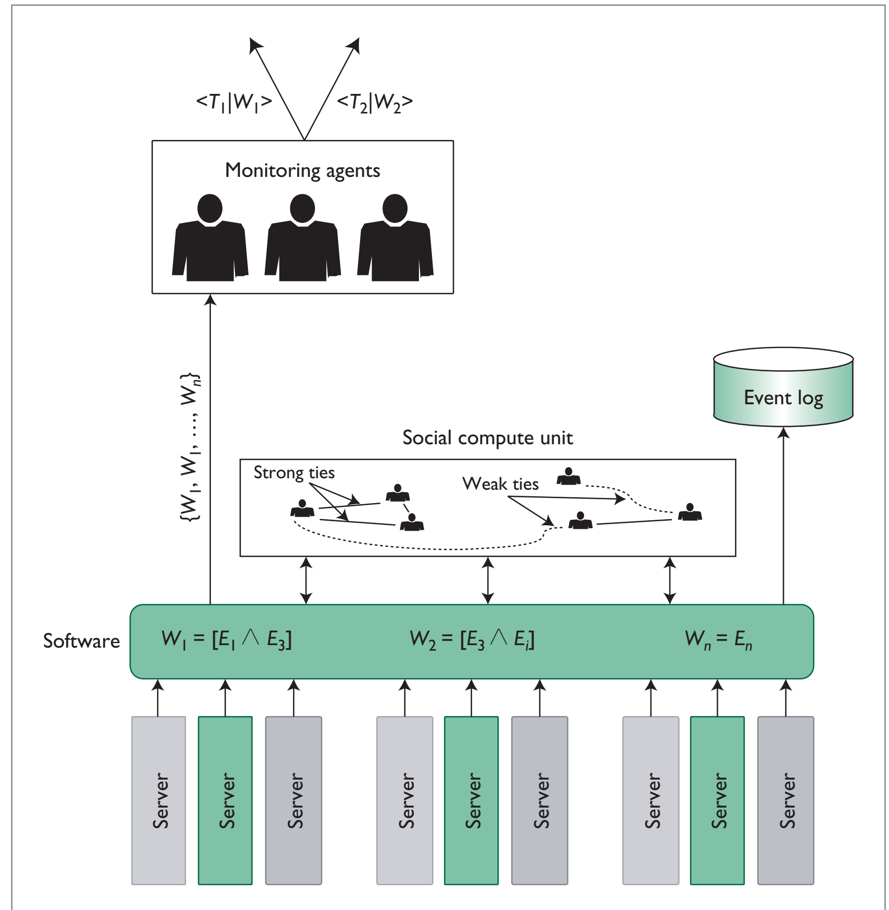

Figure 1. IT management monitoring. A service provider manages system operations by monitoring events directly or through software that automatically evaluates the events and sends out warnings as required. A social compute unit is a team of experts that know both how to interpret events and how to configure the software.

Example Scenario

Consider the following simplified example from IT management. An IT service provider that manages its clients' IT environments employs a set of human agents to monitor events emitted from various servers. The monitoring team's task is to analyze events and decide whether any indicate an incident that requires resolution. If so, the agent will issue a ticket to a system administrator who can investigate further and eventually resolve the problem. We must take several dimensions into account to assess the task's complexity:

- *Number of events.* The number of events published by a single server could be in the hundreds. Hence, we can expect thousands of events emitted by a larger server farm at any given point in time.
- *Event variability.* Event variability poses a cognitive challenge on those people receiving the events for the following reasons. First, each server with a different OS might emit a different type of event for the same problem. Second, servers with the same OS might not be standardized on the type of events they publish. Finally, we might receive events that are the result of event correlations configured on the server OS.
- *Change and growth.* Upgrading existing systems and on-boarding new ones require increasing investment to manage the environment. Change might necessitate training agents to deal with new event types and structures, and growth might require hiring new agents. Both factors can inhibit the enterprise from reaching economies of scale.

Passing events unfiltered from the systems to the agent is challenging with respect to all three dimensions. A software unit can alleviate some pain points. As Figure 1 illustrates, a software solution with pre-defined rules filters events by classifying them with respect to importance and issuing warnings to the human agents if the software detects a potential problem. This approach reduces the information content passed on to the agents from a large number of events to a smaller number of warnings. The complexity of dealing with variability is furthermore programmatically subsumed by the rules.

The change and growth aspect, however, remains an issue. The humans designated to configure the software solution in our example

possess domain knowledge comprising both the event interpretation and rules implementation. We must view the complexity of the overall ecosystem of agents, servers, software, and resolution teams reacting to the tickets as an intricately connected environment that jointly delivers the service to the client in an optimal fashion. Missing in this picture is a team that brings together various aspects of domain knowledge to adjust the software as change and growth demand.

Change and growth in the server environment are hard to predict; hence, maintaining a dedicated team on standby is challenging from a cost perspective. Expecting the software vendor to provide resources that understand the event-based requirements is unrealistic. By the same token, expecting a sufficient quantity of monitoring agents to be knowledgeable about the software is unrealistic as well. Finally, in the spirit of autonomic computing, we could envision a system with insight into the entire value chain — from event generation to resolution — that learns to adapt itself by creating new rules as required. This is unrealistic not so much because of technical feasibility but owing to the systemic uncertainty inherent to many delivery environments. The event as such is only one aspect of the entire life cycle of the problem resolution. Understanding the actual resolution of an event requires us to trace the resulting ticket from creation to resolution. This information is in principle available, but it could be almost impossible to extract consistently. For all practical purposes, the software won't be able to correlate an event the system emits with the resolution of the potential problem about which the event has alerted the monitoring agents.

The SCU addresses these challenges. It consists of a loosely coupled, virtual, and nimble team of resources with skills in the problem domain (event analysis, in our example) or the system domain (configuring the filtering software). Each agent isn't a dedicated resource but is willing to invest a certain amount of time whenever the requirement comes up. For example, a monitoring agent might willingly invest spare time in improving his or her work processes and offer help defining new rules. Similarly, a resource with knowledge in the software domain might offer his or her time as well. The team's mission is to augment the configuration rules both proactively (based on their collective insight into the subject matter) and reactively (based on requirements the monitoring teams have noted). The SCU members are rewarded based on the outcome produced (for example, technical leadership or number of rules configured). An SCU requestor compiles the SCU depending on the problem domain's requirements and sources it from descriptions of individual resources' capabilities. Depending on the strength of all the SCU team's facets, we can calculate a compute power for the SCU.

The system notion of an SCU implies a structured architectural approach to integrating socially enabled work styles, which we examine next.

Social Compute Unit Features

An SCU is a cloud-like virtual construct that exists only for the time required. It has a fundamental notion of computing power, where computing is executed through socially networked humans. Additionally, an SCU enables elasticity through its interaction with the underlying problem domain. Let's elaborate on these three fundamental aspects of an SCU.

Programmability

The SCU is a construct that comes into existence only on request. The requestor could be the problem domain's business owner or the software itself. This implies that an SCU's components are both discoverable and composeable, and that the composition is descriptive yet generic enough to be discoverable based on requirements across problem domains. The SCU is thus specific enough that we can program it. Program execution isn't static as with a regular program; a certain statistical uncertainty will be associated with the generated outcome.

Compute Power

The SCU has a certain compute power that's appropriate for solving a given problem. Each requestor will always want the "best" team to solve the problem, but this can come at a cost. In the same way that we request hardware resources in a cloud based on the requirement to keep costs down, we expect a request for compute power from an SCU to be commensurate within a cost–requirement scope.

The notion of the SCU's compute power is specific to the problem domain. A team that performs well in the agent-based monitoring domain might not be suitable for a different domain. Thus, its compute power, based on its inherent skills, could be high for one domain and low for another.

At the same time, compute power will depend on the requirements. Consider a requirement to solve a given problem in 10 days for a specific domain. The best resources might not offer the time required to solve the problem in the required time frame; so, you could likely get a partial solution, but not everything. Despite forming the SCU based on the best-skilled resources, you might still end up with lower compute power.

We must keep in mind that even an SCU with a very high compute

power won't guarantee the desired outcome at 100 percent certainty. Uncertainty will always be involved, which is reflected in the compute power notion. But the SCU structure will let organizations reason about compute power, providing a risk assessment for the resolution of the tasks at hand.

Elasticity

The SCU, through its interaction with the underlying problem domain, facilitates elasticity. By elasticity, we mean the transient SCU's ability to enhance or reduce the capability of the system it assimilates with (for instance, the software in the example provided earlier). The SCU by itself doesn't have a notion of elasticity, in the same way an application isn't elastic unless enabled by a mechanism that scales it up or down as required.

Solution Design

Let's next examine an SCU's structure and behavior.

Life Cycle

An SCU goes through the following states, as Figure 2 illustrates:

- *Request* — a client requests an SCU for a specific domain.
- *Create* — the SCU is compiled and matched to the specific problem domain.
- *Assimilate* — the SCU becomes familiar with the strategic tasks and receives sufficient details about the problem domain.
- *Virtualize* — the SCU is installed on the problem domain in a two-step process. First, a social-collaboration space is provided to ensure effective communication between resources. This will reside on a cloud. Second, a test environment on the cloud is provided that represents the problem domain's system manifestation (for instance, a test instance of the software in our example). In cases in which no system component exists, this step might not be required.
- *Deploy* — the SCU is now producing results that it can deploy from the virtual environment into the physical production environment. The SCU's actual outcome might be either directly deployed or processed through an external governance process.
- *Dissolve* — the SCU is released from its task and rewarded for its work, if a measurable outcome exists that's commensurate with initial expectations.

The virtualize step in the life cycle has some interesting architectural considerations. First, the notion of a collaboration space and a test environment is important to the SCU's performance (or compute power). Second, the process of traversing the life cycle requires further thought on how to discover and request an SCU. We examine these two aspects next.

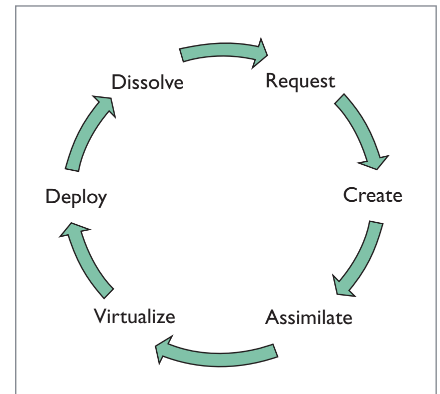

Figure 2. Social compute unit (SCU) life cycle. This life cycle indicates the construct's transient nature. After creation, the compiled unit assimilates with the problem domain until the domain owner provides a proper environment for SCU solutions tests. Once provided, the SCU can do its work, solve the problem, and deploy into production, after which the members (assuming success) will be rewarded, and the SCU dissolves.

Architecture

We propose conceptualizing an SCU as an information system. SCUs consist of a *core processing unit* comprising a network of human resources with the appropriate skill sets. The core processing unit requires a platform that facilitates communication between its nodes — for example, a social networking platform that supports the processing unit's fundamental organization.

The platform supporting the resource network can have significant impact on its performance. Imagine the platform to be a telephone — this will inhibit the team from communicating in written form and from storing information permanently. A flexible platform might provide more means for communicating but might also inhibit the SCU's compute power because it might not provide guidance as to the best way to communicate. This discussion is an important aspect of SCU design, but is out of this article's scope. The communication platform should also facilitate access to key devices, such as information repositories that might contain generic information (such as employee data) or problem-specific information.

Core Processing Unit Metadata

We envision the SCU core processing unit as following a model of distributed participatory design. The team can be distributed over various locations and business units. Team members will follow strategic directives but are sufficiently trusted to make team-based, implementable decisions. Based on the team composition's structure[3,4] and trust among team members[5] — which is automatically determined by an SCU compiler — you might require an additional control unit during ramp-up time — for example, if the team expertise isn't sufficient to make implementable decisions. The challenge in the

Table 1. Modeling elements for a social compute unit (SCU).

Modeling element	Description
Resource ID	A unique identifier for resources that participate in the SCU
Expertise	A description of expertise (for example, a key/value pair of skill area/expertise level)
Reputation	A description of reputation (excellent, very good, average, or poor)
Connectedness	A description of the resources network
Time supply	The time the resource is willing to provide
Cost	The resource's cost
Reward request	Money, time, or glory
Incentive	Money, time, or glory
Envisioned role	Leader, specialist, or moderator
Context	Context information — for example, where the resource is physically located

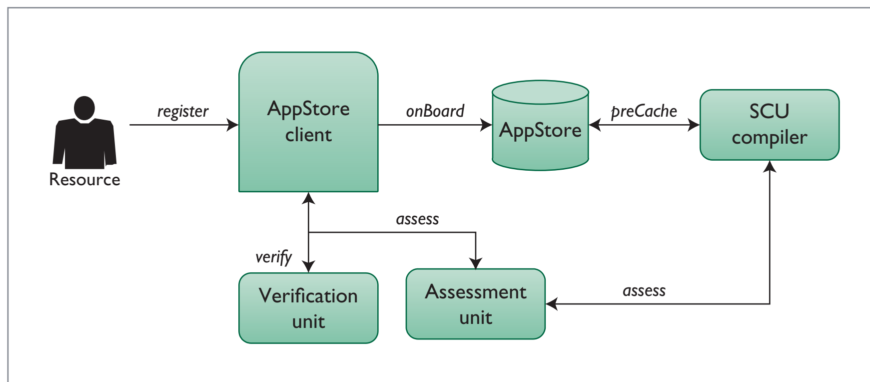

Figure 3. Resource registration. An AppStore offers a registration process that on-boards a potential social compute unit (SCU) resource. The AppStore verifies the user and assesses the resource's capability. The SCU compiler assigns the newly on-boarded resource to an SCU.

solution design is to formulate a model for SCUs that's generic enough to apply to various domain-specific contexts. Table 1 contains a first attempt to describe an SCU's modeling elements.

We must also specify the domain. We can envision leveraging domain-specific modeling or simply providing a flexible search method as an algorithm to compile SCU resources.

The AppStore

Given that we view the SCU as a structured entity, we propose to utilize an AppStore model for resource registration and solution instantiation. The idea is that the AppStore allows individuals to offer their services. The AppStore has access to the appropriate information for compiling an SCU. Figure 3 shows the registration process.

A human (resource) registers herself using an AppStore client. The data about her contains two parts: static data, such as name and employee status, which the AppStore can also utilize for security purposes, and dynamic data, which includes information such as her social network, among other things (see Table 1). The SCU compiler assesses the dynamic data the *assessment unit* creates, as well as static data that the *verification unit* verifies.

Figure 4 illustrates an SCU's instantiation process in a defined problem domain. The requestor formulates the problem using the modeling elements from Table 1 to describe a domain-specific problem as well as its underlying solution — for instance, that a team of specialists is needed who exist at a given location and have a certain amount of dedicated time and expertise, as well as envisioned roles with appropriate reputations. The SCU compiler matches these requirements to a set of possible resources (for instance, by utilizing algorithms discussed elsewhere[3–5]) and stores those in the AppStore. The *installer* is responsible for installing and configuring the appropriate SCU platform (that is, the right team with the right set of software tools), as well as a virtualized replica in a cloud infrastructure, that the team will instantiate in order to work on the given problem.

The SCU signifies a change in the way we integrate social team-based computing with workflow-type applications. The interaction patterns between the resources in an SCU determine the unit's architectural style. The interaction pattern with the problem domain, manifested as software or a system, determine the future of leveraging team-based work styles with traditional workflow systems. We propose the SCU as one framework for elaborating on behavioral and architectural styles to bring socially networked computing into the business mainstream.

References

1. D. Schall, S. Dustdar, and M.B. Blake, "Programming Human and Software-Based Web Services," *Computer*, July 2010, pp. 82–85.
2. D. Schall, H.-L.Truong, and S. Dustdar, "Unifying Human and Software Services in Web-Scale Collaborations," *IEEE*

Internet Computing, vol. 12, no. 3, 2008, pp. 62–68.

3. D. Schall and S. Dustdar, "Dynamic Context-Sensitive PageRank for Expertise Mining," *Proc. 2nd Int'l Conf. Social Informatics* (SocInfo 10), Springer, 2010, pp. 160–175.
4. C. Dorn and S. Dustdar, "Composing Near-Optimal Expert Teams: A Trade-Off between Skills and Connectivity," *Proc. 18th Int'l Conf. Cooperative Information Systems* (CoopIS 10), Springer, 2010, pp. 27–29.
5. F. Skopik, D. Schall, and S. Dustdar, "Modeling and Mining of Dynamic Trust in Complex Service-Oriented Systems," *Elsevier Information Systems J.*, vol. 35, no. 7, 2010, pp. 735–757.

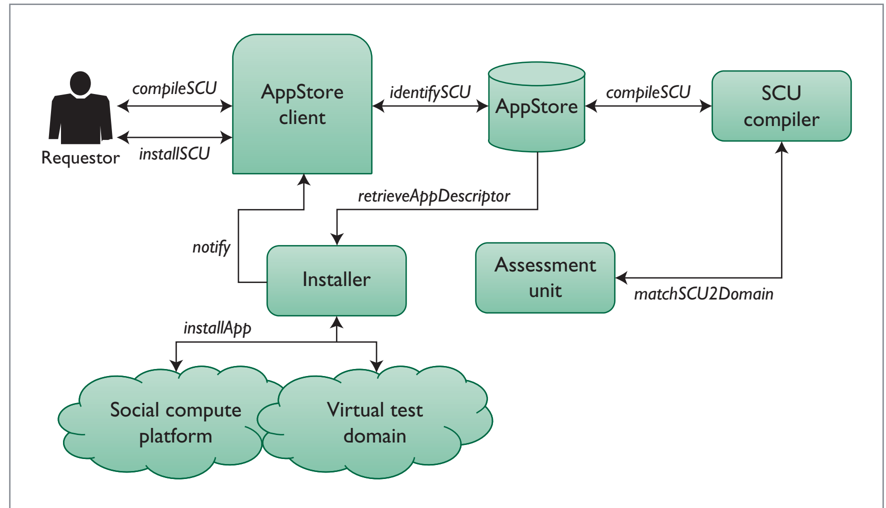

Figure 4. Instantiation of a social compute unit (SCU). An SCU can be instantiated from the AppStore, which will compile the SCU and install it, here illustrated in a model where it automatically provides a compute platform and the virtual test domain.

Schahram Dustdar is a full professor of computer science (informatics) with a focus on Internet technologies and heads the Distributed Systems Group, Institute of Information Systems, at the Vienna University of Technology (TU Wien). Dustdar is an ACM Distinguished Scientist. Contact him at dustdar@infosys.tuwien.ac.at; www.infosys.tuwien.ac.at/Staff/sd.

Kamal Bhattacharya is a senior manager at IBM Research — India, where he manages the NextGen Services team. His research targets core technical aspects of data center consolidation, cloud computing, service delivery for infrastructure, and application management, and includes efforts to investigate how new socially enabled work processes can impact services delivery in the enterprise. Bhattacharya has a PhD in theoretical physics from the Georg-August University Goettingen, Germany. Contact him at kamalb@us.ibm.com; www.research.ibm.com/people/k/kbhattacharya.

cn *Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.*

Call for Articles

IEEE Software seeks practical, readable articles that will appeal to experts and nonexperts alike. The magazine aims to deliver reliable information to software developers and managers to help them stay on top of rapid technology change.

IEEE Software

Author guidelines: www.computer.org/software/author.htm
Further details: software@computer.org
www.computer.org/software