Social Computing: Principles, Platforms, and Applications

Amit K. Chopra
University of Trento, Italy,
chopra@disi.unitn.it

***Abstract*—We present a conceptualization of social computing as the computation of social dependence among autonomous actors. Our conceptualization unifies many diverse kinds of applications such as multiparty business processes, social networks, and online discourse. We contend that the current tradition in software engineering fundamentally falls short of what is required to successfully build such applications. We propose a vision of social computing that relies exclusively on social abstractions and outline the challenges in realizing the vision.**

***Index Terms*—Actors, Autonomy, Protocols, Commitments, Trust, Distribution, Middleware, Business processes, Social networks, Argumentation**

I. Introduction

The nature of computation is changing. It is evolving from activity and data-orientation to interaction-orientation. Social networks, social cloud, e-business, virtual organizations, and so on are evidence of the shift. In such applications, autonomous social actors interact in order to exchange services and information. However, software engineering has not kept up with the ongoing shift. It remains rooted in a logically centralized perspective of systems dating back to its earliest days; it is still rooted in low-level control flow abstractions.

We understand *social computing* as the joint computation by multiple autonomous actors, whether individuals, organizations, or their software surrogates. "Joint" refers to their interactions and the social relationships that come about from the interaction, not necessarily cooperation, integration, or any other form of logical centralization. The purpose of the computation may be to carry out a multiparty business transaction, to schedule a meeting, to loan a book to a friend, to build consensus on an issue via argumentation, or globally-distributed software development itself—anything that would involve interaction among actors. We refer to applications that perform social computations as *social applications*.

Current approaches take a logically centralized view of systems: to build a system is to build a computer. Internally, the computer may well be modular in construction and distributed, but it would effectively represent only one locus of control—that of its stakeholders. Even expansive takes on software construction such as programming in the large (PiL) [1], which admitted independently built components, did not go so far as to accommodate autonomous actors as components. Recent work in architecture-based adaptation relies on centralized adaptation managers to rewire components [2]. Following Zave and Jackson [3], requirements engineering (RE) emphasizes the importance of modeling and analyzing the environment in coming up with system specifications. The notion of a system as a computer, however, remains entrenched in RE. Agent-oriented software engineering approaches are also logically centralized. Jennings [4], for example, describes a scheduling problem that is addressed by *distributing* it across intelligent agents. In short, the established ideas have served us well in building controllers: for example, operating systems, databases, transaction managers, and flight control systems. However, they fall short for building social applications, which involve interactions among social actors.

Clearly, we are already building social applications, even with current software engineering approaches. For example, online banking is a social application in which a customer interacts with one or more banks to carry out payments, deposits, and transfers. Blogs represent a social application in the sense that authors and readers express their positions and argue about them. Social networks such as Facebook and LinkedIn facilitate interactions among theirs users. However, just because we can build social applications, it does not mean we are building them the right way. Software engineering research seeks to make building applications ever easier, and the biggest leaps often come by way of new abstractions. Our principal claim is that current software engineering approaches lack the social abstractions to systematically build social applications. In fact, the social aspects of these applications are currently handled offline. For example, the contractual relationships that arise in a business interaction are understood by the interacting principals, but not by the software supporting their interaction. Lacking social abstractions, each application is built from the ground up from low-level abstractions based on control flow. This limits reusability and leads to unnecessarily complex software models and code, which in turn leads to software management headaches. If we could come up with the right social abstractions, seeming diverse social applications such as business transactions, software development, blogs, and social networks could potentially be built around the same fundamental concepts and run on a common platform that embodies the concepts.

II. The Nature of Social Computing

Social applications are studied under various guises in computer science depending on the particular aspects stressed: business processes, service-oriented computing, sociotechnical systems, virtual organizations, e-government, and so on. We

discuss three broad but diverse classes (not mere instances) of such applications, and then show that there is more commonality among them than may be readily obvious. Our aim is to exploit this commonality to inform the construction of all social applications.

Multiparty business processes: More and more business transactions are being conducted on the Web. Initially, it was the resourceful organizations that set up e-business systems; however, with advances in Web technology and the advent of online marketplaces such as eBay, even individuals with limited resources were able to start engaging each other in business transactions. The original motivation behind service-oriented computing was a common set of abstractions, protocols, and platforms for enabling such multiparty business transactions and processes.

Social networks: Another orthogonal but more recent direction in the evolution of the Web is social networks where actors are the nodes of the network and social relationships among actors the links. Users are increasingly reliant on social networks to organize social and business activities, not just to publish information. Social networks are being used in money lending, to facilitate book-sharing and carpooling, to help travelers find hosts to stay with, and in myriad other ways.

Online discourse: The Web supports semi-structured social interactions among actors via online forums and blogs (including microblogs, such as Twitter). Nowadays, writing comments to blogs posts and news items is increasingly commonplace. Often in response to a post, people argue vehemently for their own viewpoint. Organizations, including governments, are increasingly using blogs to solicit opinions and ideas from their audiences. Researchers are increasingly looking at ideas from discourse theory, especially argumentation to capture these interactions in a more meaningful and structured way. Issue-based information systems (IBIS), a traditional application area in computer science, seeks to apply argumentation towards capturing knowledge in software development lifecycles.

Many of these social applications are primarily built on top of the Web. However, the Web was conceptualized essentially as a distributed, resource-oriented, hyperlinked database. Consequently, it offers only correspondingly low-level database abstractions (the HTTP primitives). It lacks high-level social abstractions for easily building social applications. Some social applications run on their own specialized layer that runs on top of the Web (or other suitable infrastructure). For instance, business applications are often built on top of the WS-* platform. However, WS-* lacks any social abstractions whatsoever. Consider especially that business workflows, a widely prevalent way of programming such applications, are not even actor-centric—they are activity and flow-centric. Social applications such as Facebook and LinkedIn are conceptually similar; however, they share no common abstractions, application programmer's interface (API), and infrastructure except at the level of the Web. Moreover, the two networks remain noninteroperable even though a contact on LinkedIn could well be a friend on Facebook. Naturally, there is always the question of whether organizations want interoperability. But suppose they wanted it. Do we have the platform to support their interoperability? Discourse on the Web is only recently beginning to receive attention. Currently, we observe little structure beyond posts and comment trees (comments to comments and so on), although some recent efforts such as debategraph.org have begun to support a richer argumentative structure.

What is the essence of being social? What unifies these applications conceptually? Our fundamental insight is that all social applications can potentially be built from the same social abstractions. The idea of *social dependence* among actors is what lies at the heart of all social applications. Manufacturers depend on suppliers for parts. Patients depend on laboratories to deliver accurate test results. When a bidder wins an auction on eBay, the seller depends on him to make the payment. When a user announces a party at his home on Facebook, the invitees depend on him for hosting the party. When a person agrees to host a visitor, the visitors depend on him for accommodation. When a person argues in an online forum that globally glacier volume was decreasing at an annual rate of 5% or when a stakeholder argues for the implementation of a certain requirement, they are both beholden to other participants for the truth of their claims (regardless of whether the others believe them).

Social dependencies arise naturally in situations that involve interactions among actors. In fact, the dependence is social because it is grounded in interaction. Social dependence is a fundamental unifying notion, a semantic notion that cuts across applications. Social computation is the joint computation of the actors, not in the sense of common goals or intentions, but in the sense of the evolution of the network of the social dependencies among actors. For example, when a supplier delivers the parts, the manufacturers dependency on the supplier is satisfied, but the supplier may now depend on the manufacturer for payment. Popular social networks emphasize relationships such as friend, colleague, family, coauthor, and so on, which are application-specific, and in that sense, relatively arbitrary. We posit that social dependence lies at the base of even these relationships. For example, one depends on coauthors to disseminate joint work, on grandparents to babysit the children, and on friends to not forward pictures and information, such as telephone numbers and addresses, that they are privy to. Naturally, the idea of social dependence leads to the idea of social networks where the fundamental links between actors are social dependencies. Other kinds of relationships, such as coauthor, could presumably be built on top of such dependencies.

III. Challenges

To put it in a nutshell, the problem essentially is that programming any social application today is akin to writing

Emacs in microprocessor assembly language or programming banking applications over TCP/IP sockets. There exist no common social abstractions, languages, principled methodologies, and infrastructure for building and running them. This shortcoming will only become more evident and its effects more severely felt as we see a merging of what were traditionally considered different classes of applications. For example, the emerging area of social cloud seeks to exploit business services on social networks; social networks are increasingly being used to carry out political discourse; and sociotechnical systems are being applied to coordinate autonomous organizations.

Social dependence networks would represent logically, in a uniform manner, all kinds of social applications that are today understood and tackled in very diverse manners, including the ones discussed above. If we could understand the fundamental nature and forms of social dependence, we could build a social platform that would support interoperability and a corresponding API that would make programming social applications far easier than it is today—analogous to how TCP/IP promoted interoperability between computers and made programming network applications easier via a socket-based API.

We identify the following challenges in coming up with a principled methodology for social applications.

Identify and formalize the social abstractions. What is the nature of social dependence? How can we represent, compute, and reason about social dependence? What would constitute the fundamental patterns of social dependence from which composite patterns could then be built? The questions above stress not only technical rigor but also generality—via social dependencies we should be able to capture and reason about a rich variety of patterns, from relatively simple ones such as those that arise in scheduling an appointment to those that arise in argumentation and business contracts.

Devise a social application language. The language would express the architecture of an application in terms of roles and social abstractions. What is the set of features such a language should support for it to be expressive enough for a wide range of applications? For example, we should be able to encode simple business contracts, service-level agreements, quality constraints, delegation, temporal constraints on events, timeouts, and compensation. Can we express all of these using solely dependencies without resorting to lower-level abstractions?

Identify the software engineering principles. What are the principal specification artifacts of social applications? What are the software engineering principles and methodologies for building these artifacts?

Build a social middleware and API. The middleware represents the social layer—a platform—on which all social applications run, much like how socket applications run over TCP/IP and Web services run over HTTP (or WS-*). However, unlike them both, the middleware *understands* social abstractions. What are the basic platform services? What are the protocols for providing those services? How does the platform compute dependencies and maintain interoperability among actors? What is the basic API that provides access to platform services? How can we ensure that the API is extensible so we simplify programming high-level patterns?

IV. Approach

Social applications are necessarily embedded in the real world, in the sense that the actors derive their autonomy from the autonomy of their principals and social dependencies are real world relations that evolve due to the communication among actors. Two promising candidates for capturing the notion of dependence are social commitment [5], [6] and social trust [7]. For example, when a supplier commits to a manufacturer to deliver parts, it means the manufacturer can depend on the supplier for parts. In this sense, a commitment is essentially an elemental contract. The notion of dialectical commitment, a special kind of social commitments, can be applied to model the content of online discourse. The concept of social trust is relatively new; unlike cognitive trust, it emphasizes the architectural specification of social applications by capturing trust relationships among roles. We conjecture that social applications may be specified primarily in terms of trust relationships between actors; operationally, however, trust relationships may need to be backed up by social commitments. For example, a healthcare system architect may specify that patients trust the local government for scheduling appointments with doctors; however, operationally, patients would expect that this trust be backed up by the appropriate commitments and contracts from the involved parties.

We need to understand the core software engineering principles such as modularity, abstraction, encapsulation, and separation of concerns anew for social applications. Consider modularity, for instance. Because each actor is autonomous, actors would form the essential unit of modularity. Perhaps this in itself may not appear surprising when stated as such, but consider that service-oriented approaches that rely on business workflows completely ignore this principle. Plus consider the implications of modularity by actors for internally complex actors such as organizations: because organizations themselves consist of actors which may themselves be complex, one would have no recourse but to model the interactions among the actors all the way down.

Clearly, application-level protocols (as against the middleware protocols), as specifications of interaction, will form the cornerstone of our approach. For example, Facebook would be one application-level protocol in our approach; eBay would be another. Traditional languages for specifying protocols, for example UML interactions diagrams, statecharts, and so on are too low-level for social applications. Instead, protocols must be specified in terms of how communications affect the social dependencies; in other words, communications must be mapped to the social abstractions supported by the middleware.

The role of the middleware is to operationalize the application-level protocols in terms of the elementary protocols. The social dependencies between actors would evolve

by enacting the elementary protocols. An important role of the middleware would be to maintain interoperability. The middleware would also support protocols for discovering actors, for example, to discover auctioneers that have a good reputation or experts on a particular subject. We expect techniques from referral networks to play a key role here. The middleware would itself run on commonly available infrastructure such as HTTP or enterprise services buses (ESBs).

To demonstrate the advantages of our approach, we would need to build prototype applications that incorporate business interactions, argumentation, and social networks. Two that we are especially interested in are software project management (visualizing software development as an interactive activity among stakeholders) and a personal information system that helps users keep track of their commitments (similar to a calendar, only far more versatile).

V. Conclusions

Our vision of social computing differs substantially from some recent ideas in this area. Some recent manifestos and funding programs emphasize the notion of powerful social computers that can engage humans (for example, as in crowd computing) as problem-solving elements and take into account laws and social conventions [8]. Social computing, as we envisage, is complementary to the notion of a social computer. In our framework, the social computer would simply be one single actor that runs on top of the social middleware. Imagine a network of social computers. Our vision of social computing (via the social language) would be indispensable to the application that the computers collectively represent (for example, a virtual organization for resource sharing). Our vision would give the network teeth (via the middleware). Our vision would make programming social computers easy (via the social API).

By social computing, we also do not refer to social approaches such as collaborative filtering and tagging to answer queries. In those approaches, while input from multiple actors is gathered, there is one centralized computer that mines user inputs to answer queries; there is no direct interaction between actors. The algorithms for collaborative filtering and tagging could be thought of a single social computer. For the same reason, running Google's PageRank is not doing social computing. The distinction between a social computer and social computing is analogous to the distinction between algorithm and interaction.

Social computing is not about the joint *goals* or *intentions* of actors, abstractions that have been used to capture the mental states of actors and have proved influential in distributed AI (especially via speech act theory) and RE. In fact, our vision of social computing makes no assumptions whatsoever about actor state, rationale, or plans. Social computing is about the public, the observable. At any given instant, the network of social dependencies reflects the observable state of the application. And therefore social dependencies cannot be formulated in terms of mentalist notions (we refer interested readers to [9], [10]).

The principal breakthroughs from our vision would be (1) a paradigmatic shift in the way we think about, model, and engineer social applications, and (2) a radically different platform on which to run these applications. In contrast to existing platforms, the social platform would understand (and compute) high-level social abstractions and patterns thereof. Our earliest platforms for building applications were operating systems and databases; TCP/IP emerged as the platform for network applications; now, we are largely building applications on top of the Web, but there does not exist yet a single social platform on which to run social applications. Yes, we have instances of social applications, but today they run directly on the Web, not on any social platform. Our proposed social platform will address this mismatch. The platform will potentially prove to be as important to the future of social applications as the Web has been for the distribution of information. Web 2.0 has proved to be empowering for individuals and small businesses. A clear conceptual understanding of social applications and an infrastructure that vastly simplifies building them will prove to be more empowering by orders of magnitude.

The realization of the objectives of this vision will lead to common conceptual bases for different disciplines such as service-oriented computing, sociotechnical systems, and social networks and spur fundamentally new directions of research.

Acknowledgments. *This vision owes a substantial intellectual debt to Munindar Singh. Work with Fabiano Dalpiaz, John Mylopoulos, and Paolo Giorgini helped streamline the vision. Matteo Baldoni, Nicolas Maudet, Pınar Yolum, Michael Huhns, and Michael Jackson gave helpful comments. The research was supported by a Marie Curie Trentino award.*

References

[1] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the small," *IEEE Transactions on Software Engineering*, vol. 2, no. 2, pp. 80–86, Jun. 1976.

[2] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[3] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 1, pp. 1–30, 1997.

[4] N. R. Jennings, "On agent-based software engineering," *Artificial intelligence*, vol. 117, no. 2, pp. 277–296, 2000.

[5] M. P. Singh, "Semantical considerations on dialectical and practical commitments," in *Proceedings of the 23rd Conference on Artificial Intelligence*, 2008, pp. 176–181.

[6] N. Desai, A. K. Chopra, and M. P. Singh, "Amoeba: A methodology for modeling and evolution of cross-organizational business processes," *ACM Transactions on Software Engineering and Methodology*, vol. 19, no. 2, pp. 6:1–6:45, 2010.

[7] M. P. Singh, "Trust as dependence: A logical approach," in *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 2011, pp. 863–870.

[8] F. Giunchiglia and D. Robertson, "The social computer—Combining machine and human computation," University of Trento, Tech. Rep. DISI-10-036, 2010.

[9] M. P. Singh, "Agent communication languages: Rethinking the principles," *IEEE Computer*, vol. 31, no. 12, pp. 40–47, Dec. 1998.

[10] A. K. Chopra, A. Artikis, J. Bentahar, M. Colombetti, F. Dignum, N. Fornara, A. J. I. Jones, M. P. Singh, and P. Yolum, "Research directions in agent commmunication," *ACM Transactions on Intelligent Systems*, 2011, to appear.