Programmable Protocols for Social Networks

Roi Ronen and Oded Shmueli

Faculty of Computer Science
Technion - Israel Institute of Technology
ronenr@cs.technion.ac.il
oshmu@cs.technion.ac.il

Abstract—We study automation in social networks using protocols. We introduce a generic social protocol which addresses the basic problem of making a determination of status regarding an entity in the social network (e.g., an event such as a party, a conference, a demonstration, a commercial offer etc.).

The generic protocol can be modified into a wide variety of concrete protocols corresponding to likely usages, using programming-like and sometimes query-based features such as deadlines and forwarding privileges.

The high usefulness of the protocol is demonstrated through examples of concrete protocols, and a prototype implementation over Twitter is presented.

I. INTRODUCTION

Online social networks, such as Facebook [2], MySpace [6], Twitter [11] and LinkedIn [5], have gained unprecedented popularity. The number of users and connections in social networks continues to grow, as the networks proliferate to new target audiences and add new features.

This popularity has brought very large amounts of data into social networks [1], which are difficult and expensive to manage manually. Even the data available to a single user has become hard to manage manually, and automation features are being developed (e.g., [3], [15], [8], [9], [7], [13]). This report describes automation using protocols.

We introduce a protocol which automates the determination of status regarding an event in the social network (e.g., a party, a commercial offer, a demonstration etc.). A protocol instance is started by an initiator and unlike protocols discussed in [9] or [10], the instance does not commit or abort, and all the decisions made by participants in the instance are final. In addition, the protocols in [9] and [10] provide coordination for regretting a decision, rather than initially determining the decision.

The protocol that we introduce automates the coordination of such events. We use a generic protocol and modifiers, which are programming-like, and sometimes query-based, features which tune the generic protocol to fit particular needs.

A. Examples

(1) Consider a university reunion event. Bob, the organizer, would like to use the social network in order to organize the event, according to the following rules:
- There are two types of invitees. Alumni, and Bob’s personal friends, who are not alumni.
- Alumni may forward the invitation to other alumni. Others may not.
- Alumni who would like to attend need to RSVP at least two days before the event.
- $k$ sleep-over beds are reserved by Bob. These will be allocated, by the protocol, to the first $k$ interested guests.

In order to realize these requirements, Bob will be able to initiate a protocol instance for the reunion. The instance will have the following properties (to be defined in Section III): forwarding privileges will be given to participants satisfying a query that checks whether a participant is an alumni. Participants interested in setting their status to true (attend) have a feedback (RSVP) and deadline requirement. A resource of $k$ units is to be distributed by the protocol.

(2) The following example is the first message of a protocol instance, using which $p$ organizes a dinner with $k$ places, and the invitees determine their status (true or false) regarding the dinner. The invitation is to participants distant at most two edges from $p$ (say, satisfying query $q_2$). Each initial invitee receives $k'$ places, and may take up to two places. These conditions are conveyed in a structured message that shapes the possible subsequent protocol messages, which coordinate event. The semantics is to be defined. The message is:

$\text{(Event=dinner; Type=1;}$
\begin{itemize}
  \item Forwarding=$(q_2, \text{unlimited})$
  \item Feedback=$(p, \text{true); Resource=(k',2))$
\end{itemize}

II. MODEL

Network. We use a graph-based model for the social network. Nodes represent participants in the social network. Edges represent connections between participants. Some online social networks are undirected (allow only reciprocal connections, e.g., Facebook [2]), and some are directed (allow non-reciprocal connections, e.g., Twitter [11]). The protocol to be presented does not make a direct use of edge direction, and could be implemented for both directed and undirected models. However, since the directed model is more general and can simulate the undirected model, we choose the directed model. Following the Twitter terminology, if $(v,u)$ is an edge, we say that $u$ follows $v$.

Events. The protocol coordinates decisions regarding entities in the social networks, which we call events. An event may be a party, a demonstration, a commercial offer, a virtual

1Research partially supported by the Ministry of Science, grant 3-6472, and by ISF grant 1104/05.
event, etc. We assume that decisions regarding distinct events are not dependent on each other. Therefore, we restrict the discussion to the case in which only one event exists in the network.

**Statuses.** Each participant may determine status with respect to the event. The status is one of true or false. Nonetheless, the methods we present may be extended to a larger finite set of values beyond just two.

**Communication.** As in Twitter, participants may send messages only to their followers. However, participants may also 'tweet' tweets, whose contents is available to all participants. We use this feature in order to communicate information to followed participants.

### III. Protocol

The generic protocol has three types of messages:

- **Type 1: Initiation Message.** This type of message is sent only by an initiator to some of, or all of, its followers, and is always the first message related to an entity (also called 'event').
- **Type 2: Forwarding Message.** This type of message may be sent by a participant, who received a type 1 or 2 message in a protocol instance, to its followers.
- **Type 3: Feedback Message.** Feedback messages may be sent after the receipt of a message m or types 1 or 2. For simplicity, we assume that feedback is tweeted with a tag (a common Twitter practice), assuming that the participant who required feedback follows messages with the tag. A less attractive but feasible option in an undirected network is to have feedback messages passed from one participant to the other along the path, up to the addressee.

#### A. Modifiers

The generic protocol can be added modifiers, whose role is to shape the behavior of the protocol instance. Next, we introduce several modifiers.

**Forwarding Privileges.** According to the generic protocol presented above, participants can freely send messages of type 2. In many cases however, initiators (as well as others) are likely to want to limit this ability. The Forwarding modifier provides control on the privilege to forward protocol messages. A forwarding privilege is composed of two parameters. The first controls to whom the message can be forwarded. The second controls whether there is a permission to forward further or not. Table I specifies the possible values for forwarding privileges (the Forwarding modifier).

**Example.** In Bob’s protocol from Section I-A, alumni have forwarding privilege \((q, unlimited)\), where \(q\) is satisfied only by alumni. Personal friends have forwarding privilege false.

**RSVP.** A participant who sends a message, either as an initiator or as a forwarder, may require a feedback message from the recipient when the recipient changes status. For this, we use feedback messages (type 3). A feedback requirement may be such that only participants satisfying a query \(q\) are required to send feedback. Feedback messages are sent to the participant who required them. Table II specifies the possible values for the Feedback modifier.

**Deadlines.** A message with a feedback requirement may have a deadline, by which the receiver has to determine a status w.r.t. the entity under discussion. The Deadline modifier’s value is this point in time. If this time has passed before status was determined, the participant cannot determine status w.r.t. this entity. **Example.** In Bob’s protocol, the feedback modifier is \((true, Bob)\), and the value for the deadline modifier is set to two days before the event.

**Using a Limited Resource.** Protocols may have the functionality of letting participants use a communal resource (such as tickets, carpool seats etc.). The Resource modifier is a two-tuple \((k, l)\) where \(k\) is the total amount of available resource and \(l\) is the maximum amount that one participant can take. For example, a limit of two may be appropriate for events where participants are likely to show up with their spouses. If participants do not exhaust a resource and do not want to forward it, they may return the resource by sending a type 3 message to any participant on the path from the initiator to themselves, including the amount of resource returned. A resource can be negative, for example when tasks or payments are being allocated. Note that modifying the protocol to handle a resource can be in conjunction with a regular feedback requirement. Feedback messages related to the resource return resources to one of the forwarders (indicating the returned amount). Feedback messages related to RSVP requirements are sent, as usual, on a status change.

**Example.** In Bob’s protocol, the resource modifier is set to \((k, 1)\).
B. Politeness

Effective usage of protocols requires that participants do not forward a privilege or a resource that is not available. Privileges and resources can therefore not be increased when forwarded. Abidance by this is called politeness. Next, we discuss how politeness is reflected in view of the modifiers.

Forwarding Politeness. A participant \( p \) who is allowed to forward can not give forwarding privilleges that \( p \) does not have. In general, \( p \) may replace unlimited by limited, true by query or false, and query by false. In addition, \( p \) could replace privilege with query \( q \) by a query \( q' \) where \( q' \) is contained in \( q \) (see [14] for definition of query containment).

RSVP Politeness. A forwarder, say \( p \), of a message without a feedback requirement may add a feedback requirement with \( p \) as the feedback recipient. If a message includes a feedback requirement (say, to \( q \) \( p \) can not lift the requirement, but may replace \( q \) by \( p \) as the feedback recipient. In this case, \( p \) is responsible for the feedback to \( q \) and the next recipient is responsible for the feedback to \( p \). Here too, forwarders may replace a query \( q \) with \( q' \), if \( q \) is contained in \( q' \) (i.e., \( q' \) forces 'more' feedbacks).

Deadline Politeness. A forwarder may add a deadline to a protocol message which was received without a deadline. If such a message already has a deadline \( d \), the forwarder may set a deadline which is not later than \( d \).

Resource Politeness. A participant receiving a message with a resource modifier value of \((k,l)\) may forward the message with resource values \((k',l')\) where \( k' \leq k \) and \( l' \leq l \).

IV. Usage Examples

We demonstrate the usefulness of our methods by providing examples for the range of possibilities that they embody. Entities in the social network are the objects regarding which participants make decisions (typically events). Their ids are henceforth denoted as \( e_i \). In the following examples, we give values to modifiers and briefly discuss the resulting protocol's properties. We omit a formal definition of the self-explanatory syntax.

Meeting for two. \( p \) sends the following initiation message to a single participant in order to arrange a meeting only for the two participants. A type 3 message with an RSVP is required when the invitee changes status w.r.t to \( e_0 \):

\[
\text{(Entity}=e_0; \text{Type}=1; \text{Forwarding}=false; \text{Feedback}=(true,p))
\]

Meeting with deadline. Here, \( p \) sends a non-forwardable message to a few participants, with a deadline requirement:

\[
\text{(Entity}=e_1; \text{Type}=1; \text{Forwarding}=false; \text{Feedback}=(true,p); \text{Deadline}=12:30)
\]

Setting Forwarding=\(\text{true}, unlimited\) would give the invitee an option to forward the message to anyone.

A formal party. \( p \) organizes a party to which every invitee has to come with exactly one partner. The partner can not invite anyone:

\[
\text{(Entity}=e_2; \text{Type}=1; \text{Forwarding}=\text{(true, limited)}; \text{Feedback}=(true,p); \text{Deadline}=13:00)
\]

Following is a subsequent type 2 message, sent by an invitee \( p' \), which abides by the politeness rules. Forwarding is not allowed, the deadline is 11:00 rather than 13:00, and the feedback is to be sent to \( p' \):

\[
\text{(Entity}=e_2; \text{Type}=2; \text{Forwarding}=false; \text{Feedback}=(true,p'); \text{Deadline}=11:00)
\]

Upon receiving a feedback from another participant, \( p' \) may send a type 3 message with status \( \text{true} \) to \( p \):

\[
\text{(Entity}=e_2; \text{Type}=3, \text{to } p; \text{Status}=\text{true})
\]

A surprise party. \( p_1 \) is going to organize a surprise party for \( p_2 \). Participants must be direct friends of \( p_2 \), and \( p_2 \) is not by means to become aware of the party. Let \( q_1 \) be a query satisfied by participants directly connected to \( p_2 \) (and are not \( p_2 \)). The following is the type 1 message which starts the protocol. No RSVP necessary.

\[
\text{(Entity}=e_1; \text{Type}=1; \text{Forwarding}=(q_1, unlimited); \text{Feedback}=false)
\]

An open party. This party is open to everyone:

\[
\text{(Entity}=e_1; \text{Type}=1; \text{Forwarding}=(true, unlimited); \text{Feedback}=false)
\]

Tickets Giveaway. This message is for a ticket giveaway, in which \( p \), the initiator, gives away 3 tickets to every message recipient. Every interested participant can take up to 1 ticket. Forwarding is allowed, RSVP not required:

\[
\text{(Entity}=e_0; \text{Type}=1; \text{Forwarding}=(true, unlimited); \text{Feedback}=false; \text{Resource}=(3,1))
\]

A recipient of such a message, who took one ticket and would like to give the other two to two friends, may send each friend the following subsequent type 2 message (which respects the politeness rules):

\[
\text{(Entity}=e_2; \text{Type}=2; \text{Forwarding}=false; \text{Feedback}=false; \text{Resource}=(1,1))
\]

A feedback message for returning two tickets is:

\[
\text{(Event}=e_0; \text{Type}=3 \text{ to } p; \text{Resource}=2)
\]

A dinner party with \( k \) places. \( p \) organizes a dinner with \( k \) places. The invitation is to participants distant at most two edges from \( p \) (say, satisfying \( q_2 \)). Each initial invitee receives \( k' \) places, and may take up to two places. Note that a participant need not necessarily give the second place to someone who satisfies \( q_2 \):

\[
\text{(Event}=e_0; \text{Type}=1; \text{Forwarding}=(q_2, unlimited); \text{Feedback}=(p, true); \text{Resource}=(k',2))
\]

A. Prototype Implementation

The generic protocols are implemented using Twitter's API (through Twitter4J [12]) called from a Java code which implements protocol logics. We operate on real Twitter accounts.
Upon the receipt of a protocol message, the participant is provided a GUI using which they can forward the message, according to the values of the modifiers, as well as set their status w.r.t the event. The protocols logic is enforced.

**Example.** Marge organizes an academic workshop with 12 participants. She uses her social network in order to distribute the invitations (but without exceeding 12). She invites Bart and Lisa, and gives each 6 ‘tickets’ to the workshop, with permission to forward them, and a deadline set to July 15th (Figure 1). Bart decides to attend, tweets his decision on Twitter, takes a ticket, and forwards 5 tickets with the right to forward and a deadline set to July 10th (Figure 2). Lisa tweets a negative decision, forwards one ticket without the right to forward further, and returns 5. A Twitter account following the protocol’s participants appears in Figure 3. The protocol continues until the resource is exhausted.

**References**