Fluid: Social Norms–Based Multiagent Systems on the Web

Amit K. Chopra[1] and Munindar P. Singh[2]

[1] Lancaster University, Lancaster, UK
[2] North Carolina State University, USA
{amit.chopra@lancaster.ac.uk, mpsingh@ncsu.edu}

Abstract. A *social machine* is a Web application using which users interact flexibly and creatively to carry out social processes. Currently, social machines are realized via procedural technologies such as Web services. These approaches do not capture the social semantics at the heart of a social machine. Capturing the semantics of social processes would be crucial to enhancing user autonomy, accountability, interoperability, and decentralization.
We present Fluid, a decentralized multiagent architecture in which the semantics of a Web application is represented foremost as a social protocol that captures the applicable norms. Unlike data decentralization architectures such as Solid, Fluid decentralizes not just the data, but also the application logic. Our contributions are the following. One, we demonstrate how Fluid promotes user autonomy and introduces accountability as a counterbalance to autonomy. Two, we demonstrate how interesting sociotechnical patterns, e.g., relating to information governance and sanctions, may be captured in Fluid. Three, we demonstrate how Fluid applications may be realized using data decentralization technologies such as Solid.

1 Introduction

Berners-Lee [3] articulates a vision of *social machines* as open societies in which humans *engage creatively* with each other in social processes, leaving administrative tasks to computers. He envisions the Web (HTTP and supporting technologies and architectures) as a platform for such machines. Berners-Lee's vision has come true in some measure: many Web applications facilitate interactions among their users. We restrict attention to such Web applications, corresponding to social machines, which cover the domains of social media, e-commerce, healthcare, and e-government, among others.

Currently, a social machine is realized as a central Web service that mediates interactions between users. A crucial limitation of such an architecture is that it leaves the social and technical elements of a social machine disconnected. Specifically, a Web service provides the lowest common denominator functionality such as messaging or photo sharing (e.g., on Facebook and Twitter); however, social relationships between the users are epiphenomenal to the service: left to whatever the users make of their interaction.

User autonomy is essential to creativity and therefore crucial to Berners-Lee's vision of a social machine. However, current approaches interfere with autonomy. Instead of representing the semantics of the social process that a social machine supports and letting users interact autonomously in light of the semantics, the Web service (provider) *regiments* the social process, leaving little room for autonomous interaction between users—in effect, reflecting a procedural idealization of work—to adopt Suchman's criticism of traditional workflows [28].

Not surprisingly, the absence of a semantics of social processes hinders interoperability: users become tied to particular implementations and their APIs. Just as metadata and semantics are invaluable for linking data and services [6], so too are they essential for supporting social processes. The desired semantics should capture *social expectations* and *accountability* [13].

The foregoing motivates our research question: *How can we model and enact social machines in a semantics-driven manner that combines social and data semantics?* To answer this question, we take *Interaction-Oriented Software Engineering* (IOSE) [5] as our point of departure. What makes IOSE distinctive is that it advocates specifying a social machine in terms of a *social protocol* between users. A social protocol is framed as a set of social expectations between *roles*. Each user adopts a role in a protocol and deploys its agent as an implementation of that role. In IOSE, there is no central Web service; instead, we obtain loosely coupled agents that interoperate via a social protocol.

Our main contribution is *Fluid*, an architecture that instantiates IOSE using Web technologies. First, we specify a social machine, not as a Web service, but declaratively as a social protocol. To this end, we adopt Custard [4], a language that describes a social protocol in terms of *norms* (elaborated below) and provides a semantics that maps norms to a user's datastore. We demonstrate important norm patterns having to do with data access, privacy, and sanctions. We show a mapping from Custard specifications to RDF datastores by which the state of each norm (e.g., whether it is fulfilled or violated) is computed from facts in a datastore. Second, we show how to realize a user's agent as a Solid application [17, 25] to take advantage of Solid's data decentralization techniques. Users engage in a social machine by enacting its social protocol with the aid of their agents. Third, we demonstrate the flexibility Fluid accords users by letting each user deploy any agent that can participate in the relevant protocol. Thus, unlike data decentralization infrastructures, Fluid decentralizes not only data but also the application logic.

Section 2 motivates our approach by comparing prevalent architectures. Section 3 describes the language and concepts underlying Fluid. Section 4 describes the key elements and an implementation of Fluid on top of Solid, and demonstrates Fluid's flexibility in supporting multiple agent implementations. Section 5 summarizes our contributions, discusses relevant literature, and lists future directions.

2 Technical Motivation

We now motivate our choice of IOSE as a foundation for Fluid. We use examples from an imaginary photosharing social machine *PhotoShare*. We consider two requirements: (1) a user may authorize another user to view a photo stream and (2) a sharing user may prohibit the viewing from forwarding photos to others.

2.1 Centralized Logic and Data

A social machine is traditionally *logically centralized* on a Web service. This service is the locus of the social machine's definitive application logic and state, which its persists in a datastore. A significant ramification of this architecture, which Figure 1 illustrates, is that the service mediates interactions among users. The users have little control over the data the service stores—the entity that provides the service can and usually does exploit the data. Such challenges of information governance [1] and privacy [16, 21] have spurred work on decentralized data architectures, such as Solid.

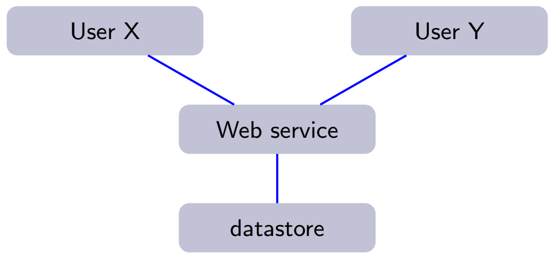

Fig. 1: A social machine implemented as a Web service.

In general, implementing an application as a Web service limits a user's autonomy to the choices made by the service provider in implementing the social machine requirements. To see this, let's assume PhotoShare is implemented as a Web service by some organization. Let's assume a user Marcy has authorized Charlie to view her photos but prohibited him from forwarding them to others. The Web service could implement this prohibition by disabling the forwarding feature for Charlie. Now suppose that Marcy is in a rural region of Italy when an earthquake hits. In this emergency situation, Charlie may wish to forward Marcy's latest photos to a rescue and relief agency (also a user of service). However, the Web service will not allow Charlie to do this.

Imagine that Charlie goes around the service by taking a screenshot of Marcy's photo and sharing it with others. Even if we trace this action to Charlie, we have no basis for claiming this is a violation of anything. If there is some textual statement that Charlie shouldn't share Marcy's photos, such a statement cannot be reasoned about computationally since it lacks a formal semantics.

In contrast, a social protocol specifies the norms (here, Marcy's prohibition on Charlie) that directly characterize Charlie's accountability. If we trace Charlie's action to him, we would know he violated the prohibition. Considerations of whether the violation itself was justified under the circumstances are subsequent to the determination of the violation and may hinge upon the user's attitudes with respect to the social interaction. For simplicity, we exclude such matters from our present scope.

2.2 Centralized Logic, Decentralized Data

Solid address challenges arising from data centralization. As Figure 2 shows, in Solid, each user has a personal datastore. Users may grant other users access certain content in their datastore via a fine-grained authorization and access control mechanism, which works together with identification and authentication mechanisms. An application (e.g., Solid social network application Timeline is implemented as a Web service that takes advantage of this decentralized data via a standardized data formats.

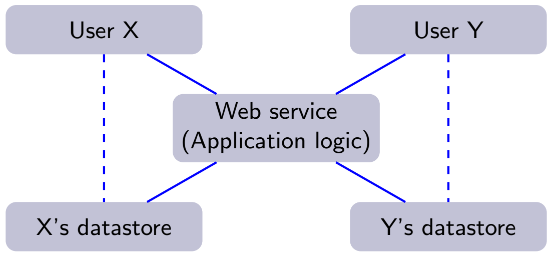

Fig. 2: A Solid application is implemented as a Web service. Users have personal datastores. The application logic comes from the Web service that executes in a user's client, and makes HTTP requests to its user's and others' datastores.

Solid tackles data decentralization, but adopts a centralized conception of application logic. Even when the logic is physically distributed across users' browsers, it is defined by a single Web service that, in effect, mediates user interactions.

Suppose we implemented PhotoShare as a Solid Web service. Marcy and Charlie would have their own datastores. Marcy's prohibition on Charlie for forwarding would be implemented via access control attributes in Marcy's datastore and corresponding logic in the Web service. In other words, the problems of autonomy and accountability would manifest themselves in Solid applications just as for vanilla Web services, as discussed in Section 2.1.

Let's consider how PhotoShare might be implemented following IOSE. The authorization to view photos and the prohibition on forwarding them would both be expressed as norms. There would be no central implementation of the norms, though. Marcy could implement the authorization for Charlie by sending photos to Charlie upon request. Alternatively, she could violate the authorization by not sending Charlie photos upon request. However, if she did violate the authorization, it would be captured in both their datastores (from the absence of a message containing the requested photo). Analogously, Charlie may receive a photo from Marcy and violate the prohibition by forwarding it to the rescue agency. The violation would be inferred from the prohibition and the event in Charlie's datastore that represents communication with the rescue agency.

3 Social Protocols

We show how a social protocol, as a set of norm specifications, may be specified and realized as abstractions over RDF stores.

3.1 Norms in Custard

We adopt Custard [4] for specifying norms for two reasons. One, Custard is an expressive (supporting complex events) specification language of social protocols that covers a wide variety of norm types. Two, Custard provides a computational semantics for protocols in terms of datastores. Although Custard was developed for relational datastores, we adapt it to RDF datastores for use in Fluid.

Custard defines four types of norms: commitment, authorization, prohibition, and power. Each norm type has a specific lifecycle that determines the state of a norm based on relevant events. We explain a commitment and its lifecycle as an example. A commitment has a debtor and a creditor, representing who is committed to, and associated with two events, namely, its antecedent and consequent. For instance, a commitment in PhotoShare is that if a poster accepts a subscription request, the poster is committed to authorizing the subscriber to view her photos. In this commitment, the poster is the debtor and the subscriber is the creditor. The antecedent is the acceptance of a subscription request and the consequent is the authorization of the subscriber.

The state of a commitment is determined according to occurrence (or lack of occurrence) of its antecedent and consequent. A commitment is detached after its creation when the antecedent occurs. If the antecedent never occurs, the commitment expires. Although we conceptually say never, occurrence of the antecedent is usually associated with a deadline. A detached commitment is discharged when its consequent occurs. If the consequent never occurs (or its deadline elapses), the commitment becomes violated. Other norm types have similar lifecycle [4] which we do not present due to lack of space.

Listing 1: A relational event schema for PhotoShare.

```
SignedUp(pID) key pID
SubscRequested(sID, pID, subscID) key subscID
SubscAccepted(sID, pID, subscID) key subscID
SubscRejected(sID, pID, subscID) key subscID
```

```
ContainerURISent(sID, pID, subscID, contURI) key subscID
PhRequested(sID, pID, subscID, reqID, phID) key reqID
PhAccessed(sID, pID, subscID, reqID, phContent) key reqID
PhForwarded(sID, oID, reqID, phContent, fID) key fID
SubscCancelled(sID, pID, subscID) key subscID
```

Formally, Custard defines a social protocol as a set of norms over an *event schema*, which defines the relevant abstract event types (e.g., antecedent and consequent of a commitment). We present a relational event schema for PhotoShare in Listing 1. Each event is specified as a relation (a set of attributes) that is annotated with a key. Each event also has a timestamp to show when it has happened, which we omit from Listing 1 for brevity. For instance, a poster signs up to PhotoShare with a poster ID (pID), which is also the key for the event. A subscription request has attributes subscriber ID (sID) and poster ID (pID), to represent who makes the subscription request to whom, and also a subscription ID (subscID) as a key. Other events are specified similarly.

Two instances of an event (specification) should not have identical values for their key attributes. However, a key in one event may appear in another event as a foreign key. For instance, subscID appears in a photo request (PhRequest) as a foreign key. These foreign keys enable correlation between events. That is, a request for a photo can be correlated with a subscription. Note that a poster can send the URI of her photo container (ContainerURISent) only once for each subscription. Otherwise, different instances of ContainerURISent would share the same subscID. However, for the same subscID there can be any number of photo request events (PhRequested), since each request has its own reqID as key.

Now, we show a simple social protocol for PhotoShare in Custard. For brevity, we elide the formal syntax of Custard [4]. We start with the commitment from a poster to a subscriber to authorize the subscriber, if her request is accepted. Listing 2 shows this commitment, SubscCommr, in Custard. The first line shows the direction of the norm. That is, the poster is committed to the subscriber. The commitment is created when the poster signs up (i.e., an instance of SignedUp). The commitment becomes detached when the poster accepts a subscription request (i.e., an instance of RequestAccepted). The commitment is discharged when the poster authorizes the subscriber by creating the authorization, as we show in Listing 3. Note that the poster must authorize the subscriber within one day of accepting the request to fulfill the commitment. Otherwise, the poster violates the commitment. This deadline is defined by the expression within the brackets in the last line.

Listing 2: Commitment to authorize an approved subscriber.

```
commit SubscCommr pID to sID
 create SignedUp
 detach RequestAccepted
 discharge created SubscAuth[0, SubscAccepted + 1]
```

Listing 3 shows SubscAuth, an authorization that states the subscriber is authorized by the poster to access her photos. It is created by the poster by sending the URI of her photo container to the subscriber (i.e., the ContainerURISent event). The subscriber detaches the authorization by making a request for a photo (i.e., PhotoRequested), and the authorization discharges when she accesses the photo (i.e., PhotoAccessed). The authorization states that access should be granted to the subscriber at most in one day after her request. The authorization expires when the subscription is canceled (i.e., SubscCancelled).

Listing 3: Authorization of an approved subscriber.

```
authorize SubscAuth sID by pID
 create ContainerURISent
 detach PhotoRequested except SubscCancelled
 discharge PhotoAccessed[PhotoRequested, PhotoRequested + 1]
```

The prohibition in Listing 4 captures the expectation of posters from subscribers about not forwarding their photos to third parties. The name of the prohibition is ForwardProh. It is created when a subscriber accesses a photo by using SubscAuth. A prohibition never discharges. However, it becomes violated if the photo is forwarded by the subscriber (i.e., PhotoForwarded).

Listing 4: Prohibition of subscribers from forwarding photos.

```
prohibit ForwardProh sID by pID
  create discharged SubscAuth
  violate PhotoForwarded
```

Once the event schema and social protocol of a social machine are defined, users can start to interact according to the social protocol. As the user's interact, instances of the events (e.g., subscription requests) occur, which are stored in users' datastores, who can observe them. For instance, when Charlie makes his request to subscribe to Marcy's photos, the appropriate instance of the subscription request event is stored in Charlie's and Marcy's datastores.

Each user can determine the states of the applicable norms from his or her datastore using Custard. For instance, Charlie can use the stored events to infer whether Marcy is committed to him for authorizing him to access her photos, and whether she fulfills or violates this commitment. Custard automatically generates the requisite queries for each state of a norm (e.g., a query for each commitment state, hence five in total for a commitment) to characterize the instantiation of a social protocol, as we show in Section 3.2.

3.2 Custard over RDF Stores

Now let us turn our attention to reasoning about norms over RDF stores. The basic idea is to store event schemas and event instances in RDF stores. The event instances explicitly represent what has happened. The stores can then be queried, in our case using SPARQL, based on each norm lifecycle event query (i.e., to determine if a norm is created, expired, detached, discharged or violated). Querying an RDF store in this manner allows us to infer the social state.

An event schema defines event types. RDF supports specifying both event types along with properties defined on them and event instances along with associated property instances.

Listing 5 declares some of PhotoShare's event types. The Custard namespace (cust) asserts Event as a class and associates a timestamp with each event. Subscription request (SubscRequested), subscription accepted (SubscAccepted) and subscription rejected (SubscRejected) are events declared in the PhotoShare (ps) namespace.

Listing 5: Event types.

```
cust:Event rdfs:subClassOf rdfs:Class
cust:Event rdfs:Property cust:Timestamp
ps:SubscRequested rdfs:subClassOf cust:Event
ps:SubscAccepted rdfs:subClassOf cust:Event
ps:SubscRejected rdfs:subClassOf cust:Event
```

Listings 6, 7, and 8 show each event type's properties. For instance, a subscription request (SubscRequested) has properties to represent who (Subscriber) makes the request to whom (Poster). RespContURI is an rdfs:Container that the poster can use to notify the subscriber about its response. A 'key' property is used to refer to a relevant datum, such as a subscription identifier referring to a specific subscription instance.

Listing 6: Attributes of SubscRequested event.

```
ps:SubscRequested rdfs:Property ps:SubscID
ps:SubscRequested rdfs:Property ps:Subscriber
ps:SubscRequested rdfs:Property ps:Poster
```

Listing 7: Attributes of SubscAccepted event.

```
ps:SubscAccepted cust:hasKey ps:SubscID
ps:SubscAccepted rdfs:Property ps:Poster
ps:SubscAccepted rdfs:Property ps:Subscriber
ps:SubscAccepted rdfs:Property ps:RequestURI
ps:SubscAccepted rdfs:Property ps:PhotoContURI
```

Listing 8: Attributes of SubscRejected event.

```
ps:SubscRejected cust:hasKey ps:SubscID
ps:SubscRejected rdfs:Property ps:Poster
ps:SubscRejected rdfs:Property ps:Subscriber
ps:SubscRejected rdfs:Property ps:RequestURI
```

Listing 9 shows an event instance (ps:001) of type 'subscription requested' stored as RDF triples. Charlie makes a request to subscribe to Marcy's photo store stream at the time indicated by the timestamp.

Listing 9: An instance of SubscRequested event.

```
ps:001 rdfs:Type ps:SubscRequested
```

```
ps:001 ps:SubscID "Subscr1"
ps:001 ps:Subscriber "Charlie"
ps:001 ps:Poster "Marcy"
ps:001 cust:Timestamp "1477222062829"
```

We now turn our attention to normative reasoning in the proposed Custard layer. Each query is generated according to Custard's semantics. However, instead of detailed semantic definitions, we demonstrate these queries via examples. The original Custard semantics is based on the relational calculus queries [4], whereas here we consider SPARQL queries. Social state inferences (e.g., that a norm is violated) are made based on SPARQL query results for event instances stored as RDF resources.

Listing 10 shows a SPARQL query for inferring whether the SubscAuth authorization, specified earlier in Listing 3, is created. All we need to check is whether the create event, ContainerURISent, has occurred and consequently we are able to infer who the authorization is by (e.g., the provider ?PID) for the benefit of whom (e.g., the subscriber ?SID) based on user variable bindings and other relevant details (e.g., the subscription identifier ?SubscID).

Listing 10: A SPARQL query generated in order to determine creation of the SubscAuth norm.

```
SELECT ?PID ?SID ?SubscID ?Time1
WHERE {
        ?ID1 type ContainerURISent.
        ?ID1 sID ?SID
        ?ID1 pID ?PID
        ?ID1 subscID ?SubscID
        ?ID1 containerID ?ContainerID
        ?ID1 timestamp ?Time1
}
```

The preceding query is simple. It can be written based on the relevant event type's properties such that the ?SubscID variable is bound to the 'container URI sent' event instance's subscriber ID property subscID and so on. The only care that needs to be taken is ensuring that variables being bound to the timestamp property value (i.e., when the event occurs) of different events have unique names in case multiple events are being queried for (i.e., we do not wish to over-constrain a query to only returning simultaneously occurring events).

The SPARQL query in Listing 11 is more complicated. It tests whether the same authorization is discharged. Relying on nested clauses, it first tests whether the discharged condition has occurred, conditional on the detach having occurred previously (the first nested statement), which is conditional on the create event having occurred before the detach event (the third nested statement). By generating SPARQL queries based on norms, complex queries in application logic do not have to be handwritten, rather we can automate inferring social abstractions.

Listing 11: A SPARQL query generated in order to determine creation of the SubscAuth norm.

```
SELECT ?SID ?PID ?SubscID ?ReqID ?PhContent ?Time5
WHERE {
  ?ID4 Type PhotoRequested.
  ?ID4 timestamp ?Time4.
  ?ID4 sID ?SID.
  ?ID4 pID ?PID.
  ?ID4 subscID ?SubscID
  ?ID4 reqID ?ReqID
  ?ID4 photoID ?PhotoID
  FILTER(?Time5 >= ?Time4. ?Time5 <= ?Time4 + 1.)
  {
    SELECT ?SID ?PID ?SubscID ?ReqID ?PhID ?Time3
    WHERE {
      ?ID3 Type PhotoRequested.
          ?ID3 timestamp ?Time4.
          ?ID3 sID ?SID.
          ?ID3 pID ?PID.
          ?ID3 subscID ?SubscID.
          ?ID3 reqID ?ReqID.
          ?ID3 phID ?PhID.
          MINUS { ?ID2 Type SubscCancelled.
            ?ID2 SID ?sID.
        ?ID2 PID ?pID.
                ?ID2 SubscID ?subscID.}
                FILTER(?Time2 >= ?Time1.)
                {
                SELECT ?SID ?PID ?SubscID ?Time1
                WHERE {
                  ?ID1 Type ContainerURISent.
                  ?ID1 sID ?SID.
                  ?ID1 pID ?PID.
                  ?ID1 subscID ?SubscID.
                  ?ID1 containerID ?ContainerID.
                  ?ID1 timestamp ?Time1.
        }
      }
    }
  }
}
```

Such generated SPARQL queries are submitted to Solid datastores in order to infer the social state based on event instances. Algorithm 1 provides a pseudocode for submitting such queries, based on the communication methodology implemented by the Solid application Dokieli. The approach is quite simple: (1) create a *promise* as an asynchronous data object that initially contains no data but promises to notify the user when it is assigned data (or notify of a failure); and (2) update the promise object based on SPARQL query results for inferring whether there is a violation.

These SPARQL queries are submitted using the Solid methodology: (1) web credentials are set as being required, meaning the browser will prompt or automatically supply a certificate in order to access the Solid datastore; and (2) the SPARQL query is submitted using HTTP Post.

Algorithm 1 Sample code for posing SPARQL queries generated by Custard in order to infer the social state from Solid datastores.

```
function GETVIOLATIONS(normName)
    return new Promise(
    function FUNCTION(resolve, reject)
        httpRequest ← new httpRequest()
        httpRequest.open('POST', userStoreURL)
        httpRequest.withCredentials = true
        ▷ Additional http request parameters should be specified.
        [...]
        httpRequest.onreadystatechange ←
        function FUNCTION(resolve, reject)
            [...]
            ▷ A function that calls the resolve and reject
              call back functions based on the respective
              success or failure of the http request.
        query ← custard.violationQuery(normName)
        httpRequest.send(query)
    )
```

4 Instantiating IOSE as Fluid

Now we present our architecture Fluid that instantiates IOSE for social machines. We first present the elements of Fluid, and then explain the implementation of agents in some detail.

4.1 Key Elements of Fluid

In Fluid, the social protocol is a refinement of a social machine's requirements. A social protocol in Fluid is specified as we explain in Section 3 using Custard syntax for norms and RDF serialization for the event schema. The protocol specification itself is a resource that can be accessed by all the users from a public protocol repository.

In Fluid, users of the social machine (X and Y in Figure 3) may play one or more roles in a social machine according to their interaction with other users. For instance, in PhotoShare, Charlie plays the subscriber role when interacting with Marcy to accesses her pictures, and the poster role when interacting with users (including Marcy) who are subscribed to view his photos.

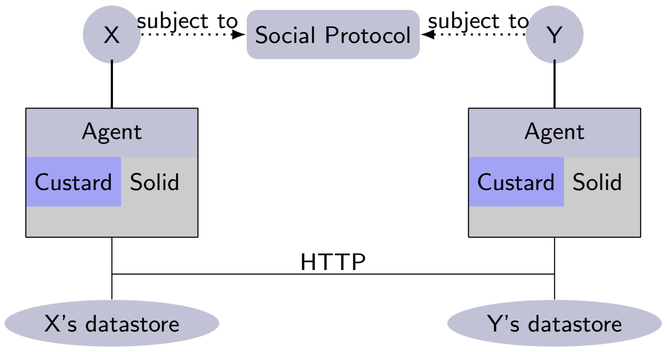

Fig. 3: Fluid decentralized social application architecture.

Users are represented by their (computational) agents. User agents are implemented on top of Solid and Custard. Solid provides the interface and middleware to access the datastores of the users. Custard provides the practical tool to reason about the social state of the social machine, which assists the agents to decide on the course of their actions and interaction with other agents. In practice, third parties would provide agent implementations that are compatible with specific roles in specific protocols.

In Fluid, each user has a personal datastore that complies with the Solid specification [17]. The schema of a datastore is determined by the user, who owns the datastore. Fluid does not enforce any constraint on the schema of a user's datastore. In the implementation level, agents communicate with the datastores through HTTP protocol, which is encapsulated by the Solid API. Authentication and access rights on resources in datastores are also handled by Solid. Although Custard provides its own API to agents to infer the social state, in the background, Custard uses the Solid API to communicate with the user's datastore. Importantly, Custard retrieves data (through GET requests) to infer the social state, but does not modify any resources on a user's datastore.

An agent may access resources in datastores of other users if they permit. In the social level, these access rights are specified in the social protocol in terms of norms. For example, when Marcy accepts Charlie's subscription request, she authorizes him to access her photos. Solid specifies how it manages access rights on physical resources using Web Access Control.

While the social aspects of interaction are specified by the social protocol, the implementation of interaction in Fluid is by creating and updating resources in datastores that correspond to event instances. For example, in PhotoShare, each party's datastore includes a designated location—a publicly writable container as specified by the Linked Data Platform. To make his subscription request, Charlie creates a new resource in Marcy's container that corresponds to the SubscRequested event. Marcy can browse her container as she wishes. Once she observes the new resource, she can interpret the event that the resource represents to reason about the event's social meaning according to the social protocol. Then she responds to Charlie in a similar way by creating a resource that corresponds to an event (e.g., accept request) in a designated container in Charlie's datastore.

4.2 Implementing Agents

Now let us look at implementing user agents. A Fluid agent is generally crafted to meet interaction requirements for a given social protocol. Moreover, a Fluid agent, both in terms of functionality and graphical interface, is implemented according to a user's role, requirements, and preferences in the social machine. For instance, when acting as a subscriber, Charlie's agent may provide functionality, such as face recognition, to identify and tag the photos that involve him. However, Marcy may not require such functionality, and can use a simple agent that shows every photo from users whose photo streams she has subscribed to.

Furthermore, the implementations of the agents are decoupled from each other: neither depends upon the internal details of the other. Instead, interoperability is achieved by defining the semantics of interaction using a social protocol. For instance, to make his subscription request to Marcy's agent, Charlie's agent does not need to know which algorithms or data structures are used by Marcy's agent. The only necessity for interoperability is the ability to initiate the event of subscription request as it is specified by the social protocol. As a result of this decoupling, Fluid achieves both application and data decentralization.

We consider two agents Charlie uses to share and subscribe to photo streams, a flexible agent (Algorithm 2) and a rigid agent (Algorithm 3), based on different requirements. Charlie's flexible agent meets the following two requirements for automatically subverting the social protocol in extraordinary cases (i.e., emergencies) and supporting Charlie's autonomy in other cases. 1. A member of a search and rescue team may request a photo to assist in an emergency, such as to identify a person's whereabouts. In this case, Charlie wishes the photo to be forwarded automatically, even though it would violate a prohibition on forwarding. 2. Charlie would generally like to send all requested photos to a confirmed subscriber, in accordance with the authorization norm. However, he may wish to edit photos before they are sent. Hence, photos are not sent automatically, instead Charlie is prompted to send them.

The pseudocode for Charlie's flexible agent is given in Algorithm 2 (for clarity we assume norms and event schemas have corresponding types in an imperative language).

Charlie's rigid agent pseudocode is given in Algorithm 3 according to different user requirements. 1. Charlie does not wish photos to be automatically forwarded should doing so violate a prohibition. Accordingly, the agent should verify whether photo forwarding would violate a norm before actually forwarding the photo. 2. Charlie wishes to automatically send all requested photos that are authorized to be sent.

It must be emphasized that social semantics are central to the agent logic presented. An authorization dialog is not presented to the user in Algorithm 2

Algorithm 2 Charlie's flexible agent implementing the PhotoShare social protocol.

```
1: function OnReceivePhotoForwardRequest(request)
2:     if request.requester ∈ authorities and
           request.circumstance = emergency then
3:         ForwardPhoto(request.photo, request.requester)
4: function UpdatedSocialState(subscAuth, photoRequested)
5:     if SubscAuth.newstate == “detach”
           and subscAuth.complEv is PhotoAccessed then
6:         AccessDialog(photoRequested.photoID,
           photoRequested.subscID)
```

Algorithm 3 Charlie's rigid agent implementing the PhotoShare social protocol.

```
1: function ForwardPhoto(photo, request)
2:     photoForwardedEvent ←
           new forwardedEvent(photo.subscriberID,
               request.otherPartyID, request.requestID,
               photo.content)
3:     if not custard.violation(photoForwardedEvent) then
4:         ▷ Low–level photo forwarding.
5: function UpdatedSocialState(subscAuth, photoRequested)
6:     if SubscAuth.newstate == “detach”
           and subscAuth.complEv is PhotoAccessed then
7:         ProvidePhoto(photoRequested.photoID,
           photoRequested.subscID)
```

(Line 6) based on low-level data changes. Instead, UI elements are created and agent actions performed based on updates to the social state. Agent logic is written in terms of high–level social abstractions, aiding in understandability and providing a clear correspondence with the social protocol the agent is implemented against.

5 Discussion

Table 1 identifies Fluid's distinctive features.

Approach	*Social semantics*	*Decentralization*
Web Service	Epiphenomenal	None
Solid	Epiphenomenal	Data
Fluid	Norms	Logic and data

Table 1: Web architectures and their characteristics.

Fluid captures the semantics of social processes via norms that apply to users. Fluid norms are not merely documentation (as contracts in natural language are); they are first-class computational abstractions that agents can reason about. Fluid norms are not rules that are executed in a rule engine; they are elements of the social state of a social machine. Crucially, users may not comply with the norms that apply to them; such a conception of norms is fundamental to autonomy and creativity. As we demonstrated in Section 4.2, users are free to implement their agents as they please. Charlie could if he wanted deploy an agent that violates the forwarding prohibition. However, although norms can be violated, they provide an implementation-independent notion of accountability, which can be applied as a standard of correct behavior in the given social machine. For example, Charlie is accountable to Marcy for not violating her prohibition regarding her photos.

In general, balancing autonomy and accountability is crucial for ensuring that a social machine would not devolve into the extremes of chaos or tyranny. In Fluid, accountability derives fundamentally from social protocols. The above notion echoes well-known intuitions from studies in political theory [12]. This is in contrast to approaches that treat deterrence (via negative sanctions) as accountability [10, 9]. Sanctioning (whether positive or negative) an accountable party is a process that is conceptually subsequent to accountability, not incorporated in its definition.

REST [11] is an architectural style for Web applications. Ciortea *et al.* [7, 8] advocate REST as a style suited to multiagent systems since agents must deal with resources in the environment. Moreover, those resources could come with interesting semantic descriptions. Extensions of Fluid to fully exploit the Web technology ecosystem would be a productive research direction.

Seneviratne [26] proposes HTTP Accountability (HTTPA) as a mechanism by providers may annotate resources with usage norms that consumers should ideally abide by but are not forced to. HTTPA supports tracking and auditing of compliance. Although the specific architectures are different, HTTPA and Fluid share similarities. It would be interesting to investigate the use of Custard in the HTTPA architecture.

Decentralized infrastructure has been a theme of growing interest in computing but much of it has focused on data, just as Solid does. This includes landmark peer-to-peer infrastructures such as Chord [27] and blockchains [29] that can be deployed to support social machines. However, these infrastructures do not address the decentralization of the social machine's application logic, as Fluid does. The logical decentralization of a social machine requires thinking in terms of a social protocol that captures the meanings of the social machine's constituent interactions. Fluid, as an instantiation of IOSE, demonstrates how social protocols may be concretely realized over the Web. Analogous instantiations can be developed over other decentralized data infrastructures. Idelberger et al. [14] report initial work on encoding blockchain applications (smart contracts) in a language that supports normative abstractions.

In security policy languages such as [19], a computer system is either obliged or prohibited from taking certain actions. These policies are programs—not social norms—that are executed by the policy engine, a distinction echoed by Polleres [22]. Fluid can be applied to specifying norms in light of which security policies are specified and thus bring sociotechnical aspects to bear. Indeed, we demonstrated how social protocols could be connected with security policies in agent implementations. Similarly, authorization framework implementations, e.g., OAuth [15], could be checked for correctness with respect to norms. Supporting sophisticated role ontologies in Fluid, as Belchior et al. [2] do for role-based access control, would be an interesting extension.

The current excitement around social machines is rooted in the potential of applying data analytics on user-generated content for solving social problems, such as earthquake prediction [24], traffic routing [20], understanding group dynamics [23], and understanding urban geographies [18]. Data analytics could benefit from Fluid's explicit representation of social norms. In particular, inference algorithms could be run on norm stores instead of lower-level data. An exciting application of such analytics would be as part of a governance feedback loop that leads to revisions of the social protocol for a social machine. In some applications, the state of a norm may be fuzzy because the underlying facts themselves are fuzzy. Approaches for fuzzy annotations for data and associated queries [30] should be valuable in extending Fluid.

Fluid supports common normative relationships such as commitment, prohibition, power, and so on. From experience though, we know that there are many more kinds of social relationships. For example, *recommends*, *likes*, *trusts* may also be seen as social relationships between users. An important direction of future work is to understand how Fluid could support idiosyncratic social relationships.

Acknowledgements We thank Akın Günay and Thomas Christopher King for valuable discussions. MPS thanks the US National Science Foundation (grant IIS-1908374) for partial support.

References

1. Au Yeung, C.M., Liccardi, I., Lu, K., Seneviratne, O., Berners-Lee, T.: Decentralization: The future of online social networking. In: W3C Workshop on the Future of Social Networking Position Papers. vol. 2, pp. 2–7 (2009)
2. Belchior, M., Schwabe, D., Parreiras, F.S.: Role-based access control for model-driven Web applications. In: Proceedings of the 12th International Conference on Web Engineering. LNCS, vol. 7387, pp. 106–120. Springer (2012)
3. Berners-Lee, T.: Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web. Harper Business, New York (1999)
4. Chopra, A.K., Singh, M.P.: Custard: Computing norm states over information stores. In: Proceedings of the 15th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1096–1105. IFAAMAS, Singapore (May 2016). https://doi.org/10.5555/2936924.2937085
5. Chopra, A.K., Singh, M.P.: From social machines to social protocols: Software engineering foundations for sociotechnical systems. In: Proceedings of the 25th International World Wide Web Conference. pp. 903–914. ACM, Montréal (Apr 2016). https://doi.org/10.1145/2872427.2883018
6. Christian Bizer, T.H., Berners-Lee, T.: Linked data–the story so far. International Journal on Semantic Web and Information Systems (IJSWIS) **5**(4), 1–22 (2009)
7. Ciortea, A., Mayer, S., Gandon, F., Boissier, O., Ricci, A., Zimmermann, A.: A decade in hindsight: The missing bridge between multi-agent systems and the World Wide Web. In: Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 1659–1663. IFAAMAS (2019)
8. Ciortea, A., Mayer, S., Michahelles, F.: Repurposing manufacturing lines on the fly with multi-agent systems for the Web of Things. In: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 813–822. IFAAMAS, Stockholm (Jul 2018)
9. Feigenbaum, J., Hendler, J., Jaggard, A.D., Weitzner, D.J., Wright, R.N.: Accountability and deterrence in online life (extended abstract). In: Proceedings of the 3rd International Web Science Conference. pp. 7:1–7:7. ACM Press, Koblenz (Jun 2011). https://doi.org/10.1145/2527031.2527043
10. Feigenbaum, J., Jaggard, A.D., Wright, R.N.: Towards a formal model of accountability. In: Proceedings of the 14th New Security Paradigms Workshop (NSPW). pp. 45–56. ACM, Marin County, California (Sep 2011). https://doi.org/10.1145/2073276.2073282
11. Fielding, R.T.: Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
12. Grant, R.W., Keohane, R.O.: Accountability and abuses of power in world politics. American Political Science Review **99**(1), 25–43 (Feb 2005). https://doi.org/10.1017/S0003055405051476
13. Hendler, J., Berners-Lee, T.: From the semantic web to social machines: A research challenge for AI on the world wide web. Artificial Intelligence **174**(2), 156–161 (2010)

14. Idelberger, F., Governatori, G., Riveret, R., Sartor, G.: Evaluation of logic-based smart contracts for blockchain systems. In: Proceedings of the 10th International RuleML Symposium. LNCS, vol. 9718, pp. 167–183. Springer (2016)
15. IETF: The OAuth 2.0 authorization framework. https://tools.ietf.org/html/rfc6749 (Oct 2012)
16. Kagal, L., Finin, T., Paolucci, M., Srinivasan, N., Sycara, K., Denker, G.: Authorization and privacy for semantic web services. IEEE Intelligent Systems **19**(4), 50–56 (Jul 2004)
17. Mansour, E., Sambra, A.V., Hawke, S., Zereba, M., Capadisli, S., Ghanem, A., Aboulnaga, A., Berners-Lee, T.: A demonstration of the Solid platform for social web applications. In: Proceedings of the 25th International Conference on World Wide Web (WWW), Companion Volume. pp. 223–226. ACM, Montreal (Apr 2016). https://doi.org/10.1145/2872518.2890529
18. Nadai, M.D., Staiano, J., Larcher, R., Sebe, N., Quercia, D., Lepri, B.: The death and life of great Italian cities: A mobile phone data perspective. In: Proceedings of the 25th International Conference on World Wide Web. pp. 413–423 (2016)
19. OASIS: eXtensible Access Control Markup Language (XACML) version 3.0 specification document (Aug 2010), http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf, OASIS Standard. Accessed 03-Feb-2022
20. Pan, B., Zheng, Y., Wilkie, D., Shahabi, C.: Crowd sensing of traffic anomalies based on human mobility and social media. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 344–353 (2013)
21. Paradesi, S., Liccardi, I., Kagal, L., Pato, J.: A semantic framework for content-based access controls. In: International Conference on Social Computing. pp. 624–629. IEEE (2013)
22. Polleres, A.: Agreement technologies and the semantic web. In: Agreement Technologies, pp. 57–67. Springer (2013)
23. Purohit, H., Ruan, Y., Fuhry, D., Parthasarathy, S., Sheth, A.P.: On understanding the divergence of online social group discussion. In: Proceedings of the Eighth International AAAI Conference on Weblogs and Social Media (May 2014)
24. Sakaki, T., Okazaki, M., Matsuo, Y.: Earthquake shakes Twitter users: Real-time event detection by social sensors. In: Proceedings of the 19th International Conference on World Wide Web. pp. 851–860. ACM (2010)
25. Sambra, A.V., Guy, A., Capadisli, S., Greco, N.: Building decentralized applications for the social Web. In: Proceedings of the 25th International Conference Companion on World Wide Web. pp. 1033–1034. ACM, Montréal (Apr 2016). https://doi.org/10.1145/2872518.2891060
26. Seneviratne, O.W.: Accountable Systems: Enabling Appropriate Use of Information on the Web. Ph.D. thesis, Massachusetts Institute of Technology (2014)
27. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup protocol for Internet applications. IEEE/ACM Transactions on Networking **11**(1), 17–32 (2003)
28. Suchman, L.A.: Office procedure as practical action: Models of work and system design. ACM Transactions on Office Information Systems **1**(4), 320–328 (Oct 1983)
29. Swan, M.: Blockchain: Blueprint for a new economy. O'Reilly (2015)
30. Zimmermann, A., Lopes, N., Polleres, A., Straccia, U.: A general framework for representing, reasoning and querying with annotated semantic web data. Web Semantics: Science, Services and Agents on the World Wide Web **11**, 72–95 (Mar 2012)