Application Patterns for Cyber-Physical Systems

Jong-Seok Choi*, Tim McCarthy*, Maneesh Yadav*, Minyoung Kim*, Carolyn Talcott*, Eric Gressier-Soudan†
*SRI International, Menlo Park, CA, USA †CNAM (Conservatoire National des Arts et Métiers), Paris, France
Email: firstname.lastname@sri.com, eric.gressier_soudan@cnam.fr

***Abstract*—Patterns are of great interest for bootstraping complex systems such as networked cyber-physical systems (NCPS). Inspired by a new programming paradigm based on partially ordered knowledge sharing model for loosely coupled distributed computing and its implementation in our *cyber-application framework*, this paper studies how to program an NCPS and exploit the capabilities provided by the underlying framework. By identifying common patterns across a variety of use-cases — e.g., cyber-physical game, local anonymous chat, and collaborative brainstorming applications — we focus on design principles of NCPS, identifying the key abstractions, how they interact with the user and the knowledge system, how the ordering among knowledge items can be defined, and what features of knowledge dissemination are the most effective for certain scenarios.**

I. Introduction

Computer-enabled systems are becoming ubiquitous, complex and often play a critical role. Architectures are moving from isolated embedded control systems to open interactive systems that sense and affect their environment, with essential integration of the cyber and the physical, hence the term "cyber-physical" system. Networked cyber-physical systems (NCPS) are formed from distributed components of diverse capabilities that interact with an unpredictable environment. NCPS can provide complex, situation-aware, and often critical services in applications such as distributed sensing and surveillance, crisis response, self-assembling structures or systems, networked satellites and unmanned vehicles, smart buildings, assisted living, medical devices, manufacturing (including 3D printing), or distributed critical infrastructure monitoring and control. As an example, consider modern cars. They are not only concerned with controlling the operation of engine, brakes, locks, and such, but also with helping the driver be aware of surroundings (other vehicles, people, obstacles), entertainment, and with monitoring system function and tracking maintenance.

In addition, there is an explosion of small applications running on mobile devices. We can envision a future in which these applications combine and collaborate to provide powerful new functionality, not only in the realm of entertainment and social networking, but also by harnessing the underlying communication and people power for new kinds of cyber crowd sourcing tasks. In such systems people are an integral part of the cyber-physical system, affecting and being affected by interactions with the system. Facilitating and managing such interactions brings new challenges to the design and understanding of cyber-physical systems.

An NCPS needs to be open in the sense that nodes may come and go. In fact, a system may assemble 'on the fly' for a given purpose. NCPS must be reactive and maintain an overall situation, location, and time awareness that emerges from the exchange of knowledge. NCPS must deal with uncertainty and partial knowledge, and be capable of a wide spectrum of operation between autonomy and cooperation to adapt to resource constraints and disruptions in communication. How do we design, build, and understand such systems? We have developed a distributed computing model and NCPS framework for simulating, emulating, and deploying applications [11], [17], [12]. Our model and framework are based on distributed knowledge sharing, and make very few assumptions about device capabilities or the communication infrastructure. This contrasts with the traditional message-passing communication model and allows focus on capability and goals rather than an individual identity and a need for reliable point-to-point communication or expensive group communication protocols. The knowledge framework provides not only robust communications, but also mechanisms for consensus, conflict resolution, and a common language for operating on the shared state of application instances. The model enables an open architecture and simplifies providing robustness and fault tolerance (Section II).

Here we report on investigations of the use of knowledge sharing and the NCPS framework to build pervasive NCPS: complex heterogeneous networked systems with humans and (autonomous) agents in the loop. They may form spontaneously, continue over time, and participants may freely come and go. Examples include vehicular networks, mobile social networks, pervasive games, or the global network of financial markets. The investigations involved three case studies: (i) *Jigsaw*: a simple illustration of the use of distributed knowledge to build a pervasive game in which players can easily come and go. (ii) *CfChat*: a mobile pervasive system enabling emergence of local anonymous chat networks, ranking of information, and consensus formation. (iii) *Brainmap*: an application for distributed collaborative brainstorming (à la Mindmap) allowing players to jointly create and edit annotated concept maps, using the knowledge framework for conflict resolution.

The main contributions of this paper are: (i) knowledge-based patterns for building pervasive applications; (ii) integration of human interaction; and (iii) techniques for achieving diverse temporal knowledge patterns.

II. Partially Ordered Knowledge Sharing and Cyber-application Framework

Knowledge is semantically meaningful information that can be generated, stored, processed, aggregated, and communicated to other nodes. In [11], we developed a *cyber-application framework (cyber-framework in short)* [1] that enables applications to capture and exploit the semantics of knowledge. The cyber-framework provides a generic service to represent, manipulate, and share knowledge across the network under minimal assumptions about connectivity. It is the foundation for a new distributed computing paradigm for NCPS. This paradigm is based on a partially ordered knowledge-sharing model that is asynchronous, and can make explicit the abstract structure of a computation in space and time. The cyber-framework provides an API that bridges the gap between users who specify their needs and nodes that can contribute their capabilities and resources.

[1]The full version is available from `http://ncps.csl.sri.com`.

978-1-4799-0798-4/13/$31.00 ©2013 IEEE

A. Partially Ordered Knowledge Sharing

Partially ordered knowledge sharing (POKS) is based on the dual notions of *local events* and *distributed knowledge*. In this model, all local computations are event-based. There are two types of events: timed events, which can be posted to be activated at any local time in the future to schedule local action, and knowledge events, which can be posted to disseminate knowledge in the network and to respond upon receiving of a new unit of knowledge. Knowledge is an entirely application-defined concept, for which equivalence and ordering relations are defined to resolve conflicts or avoid inconsistencies. In the POKS model, each node uses some of its storage as a *knowledge base* that serves as a network cache to enable opportunistic knowledge sharing in a disruptive environment. With the knowledge base, each node (i) has its own view of the distributed state of knowledge (eventual consistency[2]), (ii) performs computations based on the knowledge content or its type (no obligation to use all types of knowledge), and (iii) emits new knowledge.

An application-specific partial order $\leq$ on all knowledge items can be defined together with its induced equivalence relation as in [11]. We refer to $\leq$ as the *subsumption order* given that the intuitive meaning of $K \leq K'$ is that K' contains at least the information contained in K. With this interpretation, the induced equivalence $K \equiv K'$, defined as $K \leq K'$ and $K \geq K'$, means that K and K' have the same semantics, even if they are represented in different ways. In this situation, that is, if $K \equiv K'$, the knowledge-sharing model may (but does not have to) discard K' without delivering it to the application, if K has already been delivered. In addition to $\leq$, we assume an application-specific strict partial order $\prec$ that is compatible with $\leq$ and we refer to it as the *replacement order*, with the intuition that $K \prec K'$ means that K' replaces/overwrites K; hence, if K has not been delivered yet to the application, the knowledge-sharing model may (but does not have to) discard it, if K' has already been delivered.

POKS is of key importance for scalable implementations, because its in-network replacement of inferior knowledge decreases the communication overhead by discarding information in a semantically meaningful way and limiting the amount of knowledge that needs to be stored at each node. For example, population-based meta-heuristic optimization algorithms can encode their locally optimal solution (i.e., population) as knowledge and disseminate it [10]. The solution quality of each population can define the ordering on knowledge. Each node maintains the knowledge of the highest order among which it receives. As another example, a logical theory can be augmented by a partial ordering on facts and goals to describe declarative control of NCPS as in our self-organizing robots case study [12].

B. Cyber-application Framework

The cyber-framework provides network caching and in-network processing mechanisms to implement a distributed computing model based on POKS. From a programming perspective, it provides communication primitives that can shield the applications from the complexities of dealing with dynamic topologies, delays/disruptions, and failures of all kinds. A platform-independent and multi-level API supports both simulation and real world deployment based on the same code. By

[2]POKS relaxes the consistency property in CAP (Consistency, Availability, and Partitions) theorem for distributed systems.

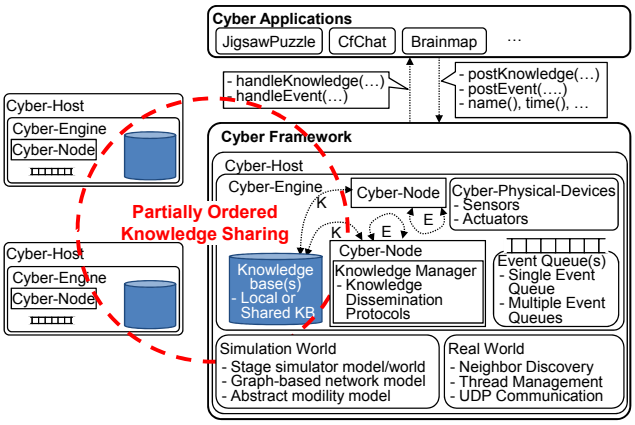


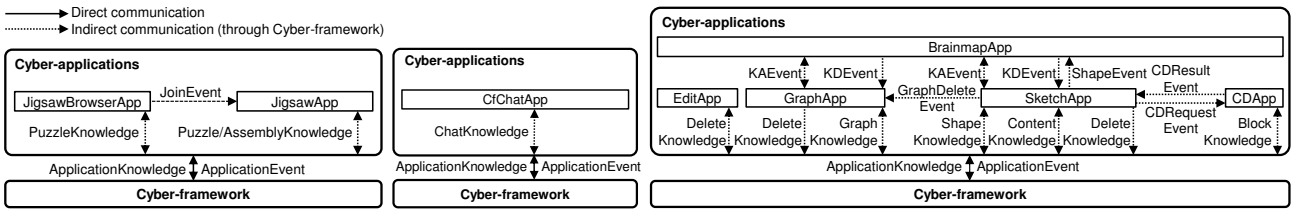
Fig. 1. Architecture of the cyber-framework: Partially ordered knowledge sharing in a locally event-driven paradigm. Knowledge is stored redundantly in local knowledge bases (KBs) providing *cyber-applications* with the abstraction of a distributed network cache. The cyber-applications have access to their cyber-node's name and local time via the API. A cyber-node provides services to post and handle events/knowledge. Details can be found in [11].

preserving the event-based structure, the API also facilitates a modular design that is easy to understand and analyze. In this paper, we aim to identify typical application patterns that exploit the capabilities of the architecture and the underlying computing model to solve problems of satisfying high-level global objectives, such as completing puzzles cooperatively, anonymous chatting, or collaborative brainstorming.

As depicted in Figure 1, the cyber-framework supports implementations with simulated and real environment including the network (e.g., wireless/wired) and cyber-physical-devices (e.g., sensors, actuators) that can be geographically scattered. The knowledge dissemination component implements specific strategies (e.g., deterministic flooding, probabilistic reflection) to propagate knowledge on top of the underlying physical network layer. The knowledge dissemination protocols make essential use of the partial order defined on knowledge, by replacing and hence discarding a unit of knowledge whenever a unit that is higher in the ordering is received. Note that what kind of knowledge that should be disseminated and how often it should be disseminated depends on the specific cyber-application and its objectives, which we will exemplify with concrete use-cases in Section III.

Each *cyber-application* runs on a *cyber-node* that is the smallest managed computational resource. Cyber-nodes can have attached *cyber-physical devices* to observe or control the environment. For example, a cyber-application for anonymous chatting among users in proximity needs to have a GPS or wireless signal strength sensor to determine situational information. Cyber-nodes form a hierarchy with a *cyber-engine* and a *cyber-host* that corresponds to a specific process and a machine on which cyber-applications are running, respectively. The neighborhood (i.e., the set of cyber-engines that are currently reachable) is maintained by each cyber-engine. The knowledge base can be shared at the cyber-engine granularity to make the knowledge available to all cyber-nodes within a cyber-engine. An event queue exists per cyber-engine to process events, which can also be parallelized using a shared queue associated with a thread pool.

By using the *cyber-API*, the interface that is provided by the cyber-framework, users can program set-up code to instantiate, compose, and configure cyber-node/engine/host and



(a) (b) (c)

Fig. 2. The architecture of three use-cases: we omitted UI (user interface) components to focus on the interaction among cyber-applications via events and knowledge. (a) *Jigsaw* has a very simple architecture designed to require minimal logic. The puzzle solving and browsing components operate independently based on the same knowledge. (b) *CfChat* pipeline is relatively straightforward with the cyber-framework by forwarding knowledge directly to the CfChatApp. (c) *Brainmap* is composed of a hierarchy of cyber-applications — *BrainmapApp* is a main application for *EditApp*, *GraphApp*, *SketchApp*, and *CDApp*. Application-specific derived events and knowledge (e.g., GraphDeleteEvent, ShapeKnowledge) are posted and handled between cyber-applications through cyber-framework's communication primitives.

applications. A cyber-node provides services to post events (local to the cyber-node) and knowledge (globally shared via dissemination) through the cyber-API — *postEvent* and *postKnowledge*, respectively. The API requires applications to define the local initialization and handling functions — *handleEvent* and *handleKnowledge*.

Events and knowledge have attributes — a *creator* (i.e., the name of the creating node), a *creation time*, an application-defined *activation time* (the earliest time when they should be handled), and an *expiration time* (after which they become obsolete). Most importantly, applications need to specialize the abstract *ApplicationEvent* and *ApplicationKnowledge* classes to make use of subsumption and replacement orderings for events and knowledge by overwriting *subsumes* and *replaces*. In this paper, we use the notation of $k(c_k, ct_k, at_k, et_k, \ldots)$ to represent knowledge k with creator c_k, creation/activation/expiration times, ct_k, at_k, et_k and application-specific attributes (if any). The ordering is specified by statements of the form $O : k(\ldots) \prec k(\ldots)$ if $condition$. For example, $O : k(c_k, ct_k, \ldots) \prec k(c_k, ct'_k, \ldots)$ if $ct_k < ct'_k$ represents an ordering based on creation time (in this case, fresher knowledge replaces older one) between knowledge items that are generated by same creator c_k.

III. Use-cases

We show how common patterns of application programming can be expressed in the cyber-framework covering the following: (i) distinctive features and description of each application (ii) architecture and its implementation, and (iii) knowledge (ordering) and application-specific dissemination policy (if any) on the cyber-framework. Figure 2 depicts the overall architecture of three use-cases. Application-specific events and knowledge are posted and handled between cyber-applications. These patterns (i.e., handling and posting of knowledge and events) to program applications are detailed in the appendix. *Jigsaw*, *CfChat*, and *Brainmap* are implemented on laptops (Ubuntu 11.10/JRE 1.6.0.26), Nexus S Android phones (Gingerbread 2.3.7/Dalvik), and both platforms, respectively.

A. Jigsaw Puzzle Game

To demonstrate the ease of implementing applications on top of the cyber-framework, we implemented a simple multiplayer jigsaw puzzle game. The advantage of using the cyber-framework to develop this application is that all of the underlying knowledge management and communications are handled automatically and transparently by the framework, allowing the programmer to focus on application-specific code.

Description: The jigsaw game implements two primary features: first, the ability to manage, create, and discover jigsaw puzzles and second, the ability to make progress in solving puzzles by assembling pieces together. When the *Jigsaw* application is launched, the user is presented with a list of puzzles currently in progress (Figure 3(a)). This list is automatically populated and updated as the cyber-framework discovers local peers and shares knowledge with them. From the puzzle list, the user can either create a new puzzle or join an existing one. After joining a puzzle, the user can drag pieces around a playing field (Figure 3(b)) to form assemblies and progress towards a solution.

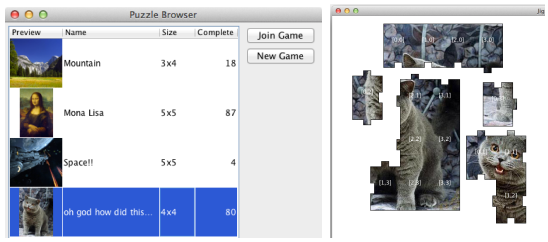
Architecture: *Jigsaw* is a knowledge-centered application, and the bulk of the features are implemented simply by defining knowledge types and their ordering, and letting the cyber-framework handle data management and sharing automatically in the background. The architecture is shown in Figure 2(a). To implement the list of puzzles, we use a kind of *ApplicationKnowledge* called a *PuzzleKnowledge* unit pk, which has the following attributes:

- *name:* title of the puzzle
- *path:* where the image file (.jpg or .png) is located
- *size:* number of rows and columns in the puzzle grid
- *uuid:* a universally unique identifier
- *cmpl:* degree of completion (%)

The JigsawBrowserApp listens for incoming *PuzzleKnowledge* items and maintains a list. An associated UI component (JigsawBrowserPanel) handles the tasks of displaying the list and of offering the user the option to create or join a puzzle as shown in Figure 3(a). When the player requests to create or join a puzzle, it posts an event which triggers the appropriate action. When a puzzle is first created, the application synthesizes a set of knowledge units consisting of all the pieces of the puzzle, each in its own assembly. These *assemblies* form the other primary data structure of the application. An *AssemblyKnowledge* unit ak is another form of *ApplicationKnowledge* and consists of:

- *loc, rot, pivot:* location (X/Y coordinates) of assembly, angle (degree) and origin of rotation on the field
- *uuid:* a pointer to the corresponding pk
- *pp:* a list of puzzle pieces included in the assembly

To start out, assemblies are placed randomly around the playing field. When the user selects to join a puzzle, a new *JigsawApp* is spawned, and it searches the knowledge base for information about which parts of the puzzle have already been assembled, building a list of assemblies to draw. JigsawApp



(a) (b)

Fig. 3. Jigsaw: (a) The puzzle browser manages the set of puzzles currently in progress, and allows users to create and join new games. (b) Puzzle in progress, with several partial assemblies.

also listens for incoming *AssemblyKnowledge* items and updates its UI component (JigsawWindow) with the latest set of assemblies to draw.

JigsawWindow handles drawing the playing field as well as handling user input to move pieces around the field. When the user finishes moving a piece, JigsawWindow tests to see if it can be merged (fitted together) with any of the other assemblies in play. For each assembly, an Affine transform is calculated based on its location, rotation, and pivot point. This transformation represents precisely how the pieces have been displaced from their original positions in the puzzle. If two assemblies have been displaced within a threshold and they contain adjacent puzzle pieces, then they can be merged and an event is posted which causes JigsawApp to post a new AssemblyKnowledge based on the union of their pieces.

Knowledge Ordering: A new unit of knowledge pk is created whenever a puzzle is created or its completion percentage changes (through the fitting together of pieces). Puzzles are considered related when they share the same $uuid$. A puzzle with higher completion replaces a puzzle with lower completion as described below, which allows information about the most complete state of a given puzzle to propagate through the network.

$$O_1^p : pk(\ldots, uuid_{pk}, cmpl_{pk}) \prec pk(\ldots, uuid_{pk}, cmpl'_{pk}) \quad \text{if } cmpl_{pk} < cmpl'_{pk}.$$

Knowledge sharing of assemblies is triggered (i.e., a new unit ak is created) whenever a new assembly is created, but not by simply moving an existing assembly around the playing field. Ordering between assemblies from the same puzzle (determined by matching $uuid$) decides whether one assembly can replace another. Replacement occurs if one assembly contains a superset of the other's pieces as below:

$$O_2^p : ak(\ldots, uuid_{ak}, pp_{ak}) \prec ak(\ldots, uuid_{ak}, pp'_{ak}) \quad \text{if } pp_{ak} \subset pp'_{ak}.$$

B. Local Anonymous Chat

Large-scale disruptions in centralized cellular communication networks due to environmental disasters or dictatorial censorship highlight the need for ad hoc network communication tools that run on ubiquitous cell phone platforms. Such tools should demonstrate efficient communication and resilience to transient network connections. Towards these goals we have designed the *CfChat* application, which runs on the Android platform and builds on the cyber-framework. User-specified "tags" (keywords) are used to organize, order, and control dissemination of knowledge. *CfChat* differs from traditional chat applications in that there is no central chat room and no need for user accounts.

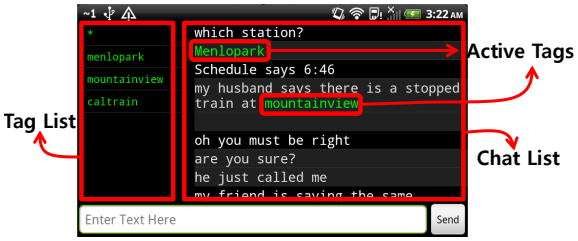


Fig. 4. CfChat: The "tag list" can be seen on the left margin of the application. Selecting the items in the tag list filters only those ChatKnowledges which contain the selected tag in their tag set. The user can "activate" a tag by long pressing the alphanumeric word with no whitespace, which highlight the word in bright green and add that string to the ChatKnowledge's set of tags and potentially to the tag list's set of tags.

Description: Although cellular networks work well in many contexts, we consider situations where spatially-local, high-volume and pseudo-anonymous communication are likely to be more useful: for instance, a disaster, such as a flood, in which users may be trapped inside houses with no cellular links, even though other cell phones may be within ad hoc networking distance. One can also consider a more mundane situation in which a group of passengers are crowded around a location and are trying to discern the precise status of a late flight. In each of these situations, we can expect that information would be transmitted with poor efficiency and poor accuracy (someone may deliberately misinform their neighbors to gain an advantage). Enabling a crowd to communicate quickly with a spatially-local neighbor would allow the efficient formation of consensus, which would be useful in these situations.

Architecture: Our chat application (Figure 4) offers the look and feel of traditional mobile messaging/chat applications, with the chat window divided into two sections: the "chat list" that displays the list of *ChatKnowledge* ck derived from ApplicationKnowledge, and the "tag list" that defines the set of active tags. ChatKnowledge ck extends ApplicationKnowledge with a message text (a string) and a set of tags (strings), which can be considered "labels" identifying categories to which a message belongs. ChatKnowledge is initially set to expire at a parameterized time (currently set to a value on the order of five minutes) but can be replaced by another ChatKnowledge with a different expiration time by tapping (for increasing) or swiping (for decreasing). The expiry of knowledge is intentional, as we expect *CfChat* to be used in environments with large volumes of irrelevant chatting, so that a general philosophy of active positive selection and passive expiry is efficient. The adjustment of expiration times and sharing of ChatKnowledge is intended to be a means of consensus: highly rated knowledge will be disseminated more frequently and take longer to expire, while poorly rated knowledge will expire quickly and have a lower likelihood of being passed on to neighbors. The basic architecture of our application is shown in Figure 2(b). The simplicity of the architecture emphasizes the ease with which loosely coupled communicating applications can be written using the knowledge-based cyber-framework.

Knowledge Ordering: We present two knowledge ordering rules for our *CfChat* that capture the notion of replacement in the context of brief, anonymous, spatially-local communication over an ad-hoc network.

While the tagging mechanism is useful for filtering knowledge and identifying common areas of interest, we only allow ChatKnowledge tag sets to grow over time since messages are

intended to be brief and transient so that it is unlikely that a user would desire to edit and delete tags from an old message that has already been sent. This notion is reflected in our first ordering rule where a given ChatKnowledge replaces another ChatKnowledge if it has the same underlying message and a superset of the tags.

$O_1^c : ck(\ldots, msg_{ck}, tags_{ck}) \prec ck(\ldots, msg_{ck}, tags'_{ck})$ if $tags_{ck} \subset tags'_{ck}$.

The ChatKnowledge in the chat list view begins to fade as it reaches its expiration time. The user can tap or swipe to replace a ChatKnowledge with a fresh ChatKnowledge that is the same except that the expiration time is increased (tap) or decreased (swipe). In order to share this change with neighbors, the newly generated ChatKnowledge must replace the existing ChatKnowledge with identical messages and tag sets: this is captured by our second ordering rule.

$O_2^c : ck(\ldots, ct_{ck}, \ldots, msg_{ck}, tags_{ck}) \prec ck(\ldots, ct'_{ck}, \ldots, msg_{ck}, tags_{ck})$ if $ct_{ck} < ct'_{ck}$.

In addition to ordering rules, we must consolidate related ChatKnowledges with identical messages and tag sets that do not contain each other, since distinct users may tag the same message in different ways. In this case we generate a new ChatKnowledge, $ck(\ldots, max(et_{ck}, et'_{ck}), msg_{ck}, tags_{ck} \cup tags'_{ck})$, with a tag set that is the union of all related tag sets and expires at the longest expiration time. Once this ChatKnowledge is posted and handled, it will be ordered against the related ChatKnowledges and replace them according to orderings.

Tag-Popularity Dissemination Policy: The tag attributes of ChatKnowledge can be used to rank them with respect to the tag-histogram of the underlying knowledge base, and use that ranking to parameterize a probability-based dissemination method in the cyber-framework. ChatKnowledge with tag sets that contain more frequent tags in the knowledge base is prioritized for dissemination. Every tag that is a member of a ChatKnowledge in the knowledge base has an associated $count$, the number of occurrences over the entire knowledge base. During each dissemination cycle, each ChatKnowledge unit ck is ranked according to the sum of the histogram counts for the i^{th} tag, $tags^i_{ck}$, in its tag set, $tags_{ck}$; $count_{ck} = \sum_i count(tags^i_{ck})$ (we do not define a deterministic method for tie-breaking equal counts). The reflection probability as defined in [16], r_{ck}, of sending a particular ChatKnowledge on to a neighbor (who is not known to have already received that ChatKnowledge) is proportional to the fraction of its sum of tag counts, over the total tag count of the entire knowledge base; $r_{ck} = \frac{count_{ck}}{\sum_{ck} count_{ck}}$, floored to a parameterized small positive value. The previous procedure is applied for any ChatKnowledge that the user herself did not create; ChatKnowledge that the user does create is distributed at every transmission duration, since every user is presumably very interested in disseminating knowledge that she creates.

C. Brainmap

To demonstrate the benefits of platform independence, we implemented a collaborative application called *Brainmap* that we deployed on heterogeneous devices including Nexus S Android phones (Gingerbread 2.3.7/Dalvik) and laptops (Ubuntu 11.10/JRE 1.6.0.26). While knowledge in the previous use-cases is only replaced by knowledge containing more information, or removed when it expires, Brainmap allows users to *edit* and *delete* knowledge. Knowledge management thus requires a more complex ordering relation. We refer to *ApplicationKnowledge* that can be edited as *EditableKnowledge*. *EditableKnowledge* can be deleted by *DeleteKnowledge*.

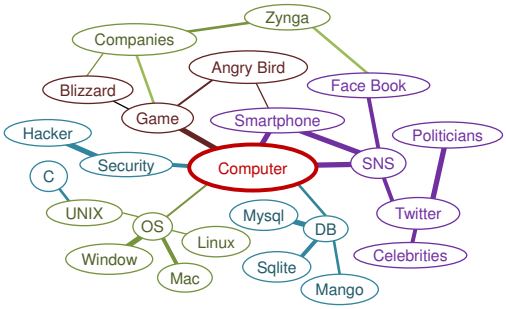


Fig. 5. Mindmap: Around the center node users can stretch their idea from a node to another node. Color of the objects (i.e., node and edge) indicates its creator. Width of the edge indicates how much the nodes connected by the edge are related.

Description: *Brainmap* is a brainstorming tool inspired by the Mindmap [18], which helps users to expand their ideas by organizing related concepts in a diagram. Consider a situation where people (who might be geographically apart) join and leave a live discussion using Mindmap. As shown in Figure 5, the brainstorming can be done by inserting and visualizing ideas from participants into the map around the center node. The map components — nodes (that can be associated with files) and edges between nodes (that can be annotated with weights) — need to be shared among users to provide a shared view of the topic (i.e., center node on a map) they are discussing.

Architecture: *BrainmapApp* is the main cyber-application that mediates the events and knowledge flow as depicted in Figure 2(c). It handles *ShapeEvent* containing geometry data created when a local user draws a diagram on a map. In Brainmap, a diagram is composed of three types of data — geometry (e.g., points and lines), content (e.g., files), and graph (e.g., nodes and edges) — that are encapsulated in ApplicationKnowledge as *ShapeKnowledge* sk, *ContentKnowledge* ck, and *GraphKnowledge* gk, respectively. BrainmapApp generates sk and corresponding gk that are posted as *KnowledgeDisseminationEvent* (KDEvent), which will be later handled by *SketchApp* and *GraphApp*, respectively. This *pairing* between sk and gk is done by assigning same $uuid$ to enable composing a node or an edge on the map from sk and gk that are separately delivered.

SketchApp generates geometry data and corresponding ck from user input. It handles sk and ck to enable pairing between them, and posts sk as *KnowledgeAcceptanceEvent* (KAEvent) that will be handled by BrainmapApp. SketchApp also generates *DeleteKnowledge* dk to replace sk and ck and posts *GraphDeleteEvent* when a local user eliminates a diagram. *GraphApp* edits and extends the graph data of a map by handling and disseminating gk. It deletes gk by posting dk when a GraphDeleteEvent is delivered. *EditApp* handles dk based on the ordering defined below to limit the volume of dk. *CDApp* implements collaborative dissemination to decrease latency by fragmenting knowledge into smaller blocks based on user-defined block size (e.g., 60KBytes). Fragmented blocks are disseminated as *BlockKnowledge* in a random order to reduce the probability of receiving the same

sequence of BlockKnowledge from multiple neighbors.

Knowledge Ordering: *EditableKnowledge — ShapeKnowledge*, *GraphKnowledge*, and *ContentKnowledge* — is created, edited, and deleted when users concurrently edit the diagram on their screen. Each *EditableKnowledge* unit ek contains a creation time ct and an identifier $uuid$, used to define the following partial order rule.

$O_1^b : ek(\ldots, ct_{ek}, \ldots, uuid_{ek}) \prec ek(\ldots, ct'_{ek}, \ldots, uuid_{ek})$
if $ct_{ek} \leq ct'_{ek}$.

Knowledge dissemination in the cyber-framework will then replace and discard all instances of ek if their creation time is earlier than that of a new unit of knowledge. This is a very typical use of a replacement ordering.

When a user deletes a node or edge in a map, *DeleteKnowledge* dk is generated to discard *EditableKnowlege* ek that is associated with the corresponding node or edge. In particular, dk contains a delete table dt with entries of ek to be deleted. This replacement is specified by a partial ordering rule

$O_2^b : ek(\ldots, ct_{ek}, \ldots, uuid_{ek}) \prec dk(\ldots, dt_{dk}, \ldots)$
if $uuid_{ek} = dt^i_{dk}.uuid$ and $ct_{ek} \leq dt^i_{dk}.ct$,

where $dt^i_{dk}.uuid$ and $dt^i_{dk}.ct$ represent $uuid$ and ct of i-th entry in dt_{dk}, respectively.

The above rule is of importance when a user leaves the network and rejoins later. Since any ek that is subject to be replaced by dk will be absorbed by the knowledge dissemination, the synchronization issue (i.e., deleted part of map is disseminated from disconnected users) can be resolved by this mechanism.

The following partial order rule is used to eliminate redundant dk units.

$O_3^b : dk(\ldots, dt_{dk}, \ldots) \prec dk(\ldots, dt'_{dk}, \ldots)$ if $dt_{dk} \subset dt'_{dk}$.

In the case of incomparable dks (i.e., $dt_{dk} \not\subset dt'_{dk}$ and $dt_{dk} \not\supset dt'_{dk}$), *EditApp* posts a new DeleteKnowledge $dk(\ldots, dt_{dk}, \ldots)$ whose dt_{dk} is $dt_{dk_1} \cup \ldots \cup dt_{dk_n}$ to replace $dk_1, \ldots, dk_n$, where n is the number of local dk, using O_3^b.

IV. Related Work

Halpern's concept of knowledge sharing [7], where facts and the state of knowledge of individual agents are expressed in modal logic, is the basis of our POKS. However, neither our POKS nor the underlying cyber-framework relies on an axiom that facts need to be true: this makes our approach feasible for real deployment. Application patterns in distributed environments can range from abstraction, aggregation, and parallelism to provide new insights into decentralized control, limited communication, use of local information, emergence of global behavior, and robustness. A mathematical theory of patterns to form ensembles in software-intensive systems with massive numbers of nodes, not necessarily homogeneous, with possibly complex interactions between nodes, is presented in [8]. The theory attempts to identify general cross-cutting concepts such as emergence, includes an abstract notion of fitness, and is used to give a precise definition of black-box adaptation, but it does not consider any operational models or algorithms that is the focus of our work.

Spontaneous near-field communication (NFC) has been investigated recently from a number of perspectives including example applications [5], [13] and protocols such as "gossiping" [4]. POKS can be implemented using such "epidemic" protocols for sending messages between nodes in range. However, our knowledge-based framework raises the level of abstraction for application development. Readily available tools and applications on widely available mobile platforms enable leveraging of a range of protocols and networking technologies for efficient implementation.

While examples of computerized jigsaw puzzles exist in abundance [1] [2], including multiplayer variants [3], the standard approach uses a centralized server. Our implementation is unique in offering decentralized drop-in/drop-out play and in its ability to split and merge in-progress game sessions based on network connectivity. The PLUG (PLay Ubiquitous Games) [6] project also aims to enhance the gaming experience without a centralized server. In these games, physical interaction is a part of the user experience. The pervasive game *PLUG The Secret of the Museum (PSM)* [6], based on the family card game design pattern, is one example. In this game, players roam around a museum to collect and exchange virtual cards, represented by RFID tags, that are located close to an artifact. The collective information store uses a causal memory model coded by hand for the game. The jigsaw case study indicates that the information exchange in such games can be represented using a knowledge ordering, and building on the cyber-framework's knowledge dissemination would simplify the design and implementation of such games.

The tags in ChatKnowledge provide an easy way to filter and organize spontaneous local chats, and to prioritize dissemination. The issue of when to trust such information is an interesting question. User reputation as a strategy for filtering untrusted information is studied in [19]. It is our expectation that the tag-based approach could be a basis for selecting useful knowledge in a stream of (potentially) noisy chat. The anonymous character of our ChatKnowledge lowers the social barrier for accurate communication, but simple schemes of user reputation are inapplicable.

Different from existing collaborative brainstorming tools such as MindMeister [14] or Mind42 [9], Brainmap supports various network configurations, including Internet and ad hoc network, and allows participants to join and leave the collaboration without disruption. Brainmap is well suited for a local network such as collaborators in a meeting room (without whiteboard), but it can also support anonymous discussion similar to the chat application. Networks of participants form and morph transparently and opportunistically thanks to the cyber-framework.

V. Concluding Remarks

This paper studies application patterns for NCPS such as knowledge aggregation, redundancy and conflict resolution, collaborative thinking, and prioritization of knowledge dissemination. These are illustrated by three use-cases: multiplayer game, local chat, and brainstorming applications. A key aspect of pervasive applications is the *convergence* properties of application knowledge. Jigsaw knowledge is persistent and monotonic in the sense that information increases. Once a puzzle is complete, eventually everyone (who is connected long enough) will see it. A player who 'joins' the Jigsaw network receives an increasingly complete view. By contrast, Chat knowledge is transitory and needs to be refreshed by a user or it will disappear. Brainmap knowledge follows yet another pattern as it concerns data structures that can be edited: created, modified, deleted in a completely asynchronous and distributed manner. The unique identifiers for knowledge instances are crucial to achieving meaningful eventual consistency.

We plan to extend this work to incorporate applications with even more physical- and social- interaction. Pervasive games ([15], [6]), which involve interaction between players with different roles as well as interaction with both physical and virtual worlds, are such examples. Patterns to express constraints in the physical world and to describe the social group behavior coordinating players are of great interest. Developing mechanisms that enable assessment of trustworthiness in applications such as *CfChat* is another topic for further research. We may also explore the integration of new cyber-physical devices with heterogeneous computation and communication capabilities such as teams of mobile robots, UAVs, and satellites. To this end, numerous technologies including location sensing, collective control of ensembles as in [8], and runtime assurance for fractionated software [17] need to be combined with application patterns.

ACKNOWLEDGMENT: National Science Foundation Grant 0932397 (A Logical Framework for Self-Optimizing Networked Cyber-Physical Systems) and Office of Naval Research Grant N00014-10-1-0365 (Principles and Foundations for Fractionated Networked Cyber-Physical Systems) are gratefully acknowledged. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

REFERENCES

[1] http://www.jigzone.com/puzzles/daily-jigsaw.

[2] http://www.jigidi.com.

[3] http://www.lunchtimers.com/game/?game=jigsaw&roomid=easy.

[4] K. Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5):8–13, 2007.

[5] F. De Rosa and M. Mecella. Designing and implementing a MANET network service interface with compact .NET on pocket PC. *.NET Technologies 2005*, page 77, 2005.

[6] E. Gressier-Soudan, R. Pellerin, and M. Simatic. *Near Field Communications Handbook*, chapter Using RFID/NFC for Pervasive Serious Games: The PLUG Experience, pages 279–304. 2011.

[7] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37:549–587, 1984.

[8] M. Hölzl and M. Wirsing. Towards a system model for ensembles. In *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *LNCS*, pages 241–261, 2011.

[9] IRIAN Solutions GmbH. Mind42. http://mind42.com, 2013.

[10] J. Kim, M. Kim, M.-O. Stehr, H. Oh, and S. Ha. A parallel and distributed meta-heuristic framework based on partially ordered knowledge sharing. *ELSEVIER Journal of Parallel and Distributed Computing (JPDC)*, 72(4):564–578, 2012.

[11] M. Kim, M.-O. Stehr, J. Kim, and S. Ha. An application framework for loosely coupled networked cyber-physical systems. In *8th IEEE Int. Conf. Embedded and Ubiquitous Computing (EUC'10), 2010*.

[12] M. Kim, M.-O. Stehr, and C. Talcott. A distributed logic for networked cyber-physical systems. *ELSEVIER Journal of Science of Computer Programming*, 2013. Available Online http://dx.doi.org/10.1016/j.scico.2013.01.011.

[13] J. Magalhães and M. Holanda. EIKO: A social mobile network for MANET. In *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on*, pages 1–5. IEEE, 2011.

[14] MeisterLabs. Mindmeister. http://www.mindmeister.com, 2013.

[15] R. Pellerin, N. Bouillot, T. Pietkiewicz, M. Wozniewski, Z. Settel, E. Gressier-Soudan, and J. Cooperstock. SoundPark : Exploring Ubiquitous Computing through a Mixed Reality Multi-player Game Experiment. In *Proc. 9th IFIP DAIS Int. Conf.*, pages 157–170, 2009.

[16] M.-O. Stehr and C. Talcott. Planning and learning algorithms for routing in disruption-tolerant networks. In *IEEE Military Communications Conference*, 2008.

[17] M.-O. Stehr, C. Talcott, J. Rushby, P. Lincoln, M. Kim, S. Cheung, and A. Poggio. Fractionated software for networked cyber-physical systems: Research directions and a long-term vision. volume 7000 of *LNCS*, pages 110–143, 2011.

[18] Wikipedia. Mindmap. http://en.wikipedia.org/wiki/Mind_map, 2013.

[19] Z. Yan and Y. Chen. AdChatRep: A reputation system for MANET chatting. In *Proc. 1st intl. symposium on From digital footprints to social and community intelligence*, pages 43–48. ACM, 2011.

VI. APPENDIX

We describe the overall structure of three use-cases as cyber-applications. We simplify computation- and UI- related components to focus on the application patterns, which are described as handling/posting of events/knowledge. In our discussion of *CfChat* and *Brainmap*, we introduce an application knowledge base to perform application-specific tasks that can be potentially computation-intensive, e.g., interacting with UI component or processing knowledge items to generate new knowledge. The application knowledge base inherits the underlying knowledge base, but contains only knowledge items of interest to a specific cyber-application and processes them. For example, handleKnowledge of *CfChat* in Algorithm VI.3 describes that an incoming ChatKnowledge is inserted to application knowledge base to be further processed. To notify the expiry of knowledge to a cyber-application, we introduce a callback *handleKnowledgeExpired* as used in Algorithm VI.3.

A. Jigsaw Puzzle Game

Algorithm VI.1: JIGSAWAPP()

```
INITIALIZE()
{
 InitializeGUI()
}
HANDLEKNOWLEDGE(k)
{
 p ← GetCurrentPuzzle()
 if(k is PuzzleKnowledge)
  then { if(k_uuid = p_uuid)
          then p ← k
 else if(k is AssemblyKnowledge)
  then { if(k_uuid = p_uuid)
          then { cmpl ← CalculateCompletion(p, k)
                 pk ← GenPuzzleKnowledge(p, cmpl)
                 POSTKNOWLEDGE(pk)
                 UpdateGUI() }
}
HANDLEEVENT(e)
{
 if(e is UserInput)
  then { if(e merges assembly)
          then { ak ← GenAssemblyKnowledge(e)
                 POSTKNOWLEDGE(ak) }
 else if(e is JoinEvent)
  then DisplayGUI()
}
```

Algorithm VI.2: JIGSAWBROWSERAPP()

```
INITIALIZE()
{
 InitializeGUI()
}
HANDLEKNOWLEDGE(k)
{
 if(k is PuzzleKnowledge)
  then UpdateGUI()
}
HANDLEEVENT(e)
{
 if(e is UserInput)
  then { if(e creates puzzle)
          then { k ← GetPuzzleKnowledge(e)
                 if(there is no JigsawApp)
                  then GenJigsawApp(k)
                 je ← GenJoinEvent(k)
                 POSTEVENT(je)
                 POSTKNOWLEDGE(k)
 else if(e joins puzzle)
          then { k ← GetPuzzleKnowledge(e)
                 if(there is no JigsawApp)
                  then GenJigsawApp(k)
                 je ← GenJoinEvent(k)
                 POSTEVENT(je)
}
```

B. Local Anonymous Chat

Algorithm VI.3: ChatApp()

```
initialize()
{
 InitializeGUI()
 e ← GenUpdateTxProbabilityEvent(delay)
 postEvent(e)
}
handleKnowledge(k)
{
 if(k is ChatKnowledge)
  then { k_c ← Copy(k)
         AddToAppKnowledgeBase(k_c)
         for ∀ChatKnowledge k_i ∈ AppKnowledgeBase
          do { if(msg_{k_i} = msg_{k_c} &&
                  tags_{k_i} ⊈ tags_{k_c} && tags_{k_c} ⊈ tags_{k_i})
               then { RemoveFromGUI(k_i)
                      k_c ← GenChatKnowledge(k_c, k_i) }
         if(k_c ≠ k)
          then { postKnowledge(k_c)
                 t ← time to warning knowledge expiry
                 e ← GenKnowledgeExpiringEvent(k_c, t)
                 postEvent(e)
                 UpdateGUI() }
}
handleEvent(e)
{
 if(e is KnowledgeExpiringEvent)
  then UpdateGUI()
 else if(e is UpdateTxProbabilityEvent)
  then { UpdateTxProbability()
         e ← GenUpdateTxProbabilityEvent(delay)
         postEvent(e) }
}
handleKnowledgeExpired(k)
{
 RemoveFromAppKnowledgeBase(k)
 UpdateGUI()
}
```

C. Brainmap

Algorithm VI.4: GraphApp()

```
handleEvent(e)
{
 if(e is KDEvent)
  then { k ← GetGraphKnowledge(e)
         postKnowledge(k) }
 else if(e is KAEvent)
  then { k ← GetGraphKnowledge(e)
         Store(k) }
 else if(e is DeleteGraphEvent)
  then { k ← GenDeleteKnowledge(e)
         postKnowledge(k) }
}
handleKnowledge(k)
{
 if(k is GraphKnowledge)
  then { k_c ← Copy(k)
         AddToAppKnowledgeBase(k_c)
         e ← GenKAEvent(k_c, BrainmapApp)
         postEvent(e) }
}
```

Algorithm VI.5: CDApp()

```
handleEvent(e)
{
 if(e is CDRequestEvent)
  then { k ← GetKnowledge(e)
         bk ← GenBlockKnowledge(k)
         for ∀ bk_i ∈ bk
          do postKnowledge(bk_i) }
}
handleKnowledge(k)
{
 if(k is BlockKnowledge)
  then { k_c ← Copy(k)
         AddToAppKnowledgeBase(k_c)
         Store(k_c)
         if(all blocks are delivered)
          then { k' ← MergeBlocks()
                 e ← GenCDResultEvent(k')
                 postEvent(e) }
}
```

Algorithm VI.6: BrainmapApp()

```
handleEvent(e)
{
 if(e is ShapeEvent)
  then { s ← GenShapeKnowledge(e)
         g ← GenGraphKnowledge(e)
         kde_s ← GenKDEvent(s, SketchApp)
         kde_g ← GenKDEvent(g, GraphApp)
         postEvent(kde_s)
         postEvent(kde_g) }
 else if(e is KAEvent)
  then { if(e has ShapeKnowledge)
          then { k ← GetShapeKnowledge(e)
                 PairingGraphKnowledge(k) }
         else if(e has GraphKnowledge)
          then { k ← GetGraphKnowledge(e)
                 PairingShapeKnowledge(k) } }
}
```

Algorithm VI.7: EditApp()

```
handleKnowledge(k)
{
 if(k is DeleteKnowledge)
  then { k_c ← Copy(k)
         AddToAppKnowledgeBase(k_c)
         for ∀DeleteKnowledge k_i ∈ AppKnowledgeBase
          do { if(k_i ≠ k_c && k_c ≠ k_i)
               then k_c ← GenDeleteKnowledge(k_c, k_i) }
         postKnowledge(k_c) }
}
```

Algorithm VI.8: SketchApp()

```
initialize()
{
 InitializeGUI()
}
handleEvent(e)
{
 if(e is UserInput)
  then { if(e edits diagram)
          then { e' ← GenShapeEvent(e)
                 postEvent(e')
                 if(e has a content)
                  then { k ← GenContentKnowledge(e)
                         postKnowledge(k) } }
         else if(e deletes diagram)
          then { k ← GetShapeKnowledge(e)
                 dk ← GenDeleteKnowledge(k)
                 e' ← GenGraphDeleteEvent(k)
                 postKnowledge(dk)
                 postEvent(e')
                 if(k has a content)
                  then { k' ← GetContentKnowledge(k)
                         dk' ← GenDeleteKnowledge(k')
                         postKnowledge(dk') }
         else
          then { e' ← GenShapeEvent(e)
                 postEvent(e') } }
 else if(e is KDEvent)
  then { k ← GetShapeKnowledge(e)
         postKnowledge(k) }
 else if(e is KAEvent)
  then { k ← GetShapeKnowledge(e)
         UpdateGUI(k) }
 else if(e is CDResultEvent)
  then { k ← GetKnowledge(e)
         handleKnowledge(k) }
}
handleKnowledge(k)
{
 if(k is ShapeKnowledge)
  then { k_c ← Copy(k)
         AddToAppKnowledgeBase(k_c)
         if(k_c has a content)
          then PairingContentKnowledge(k_c)
         else
          then { e ← GenKAEvent(k_c, BrainmapApp)
                 postEvent(e) } }
 else if(k is ContentKnowledge)
  then { k_c ← Copy(k)
         AddToAppKnowledgeBase(k_c)
         PairingShapeKnowledge(k_c) }
}
```