Synchronous Communities

John Plaice    Blanca Mancilla
School of Computer Science and Engineering
University of New South Wales
UNSW SYDNEY NSW 2052, Australia
{plaice,mancilla}@cse.unsw.edu.au
2008

Abstract
We propose the synchronous community, a set of context-aware programs held together by a single shared context that permeates all of the programs, together with a synchronous discrete semantics. The different programs may communicate point-to-point or by making changes to the shared context. With the synchronous semantics, changes to the context will be perceived by all other programs, but only in the subsequent instant. As a result, it becomes possible to define the semantics for an entire community, and even for multiple interacting communities.

1 Introduction
This position paper demonstrates how the æther of possible-worlds versioning [2] can be adapted so that a community of context-aware applications can be formed with a clear semantics for the entire system. The proposal consists of using a synchronous semantics for the entire system, with alternating phases in which 1) the æther—the shared context—merges the context deltas received from the applications, broadcasting relevant deltas back to the applications, and 2) the applications receive their respective deltas from the æther, change their respective local contexts, respond to their input and send deltas back to the æther.

The objective of this paper is to present a mechanism for building intensional communities [3], which consist of a set of applications permeated by the same shared context. The initial solution consisted of creating an æther, a data structure containing a shared context, with which different applications would register participants at different points in the context, waiting for changes thereon. This paper proposes a refinement of that model to make it act synchronously to ensure consistency across the community.

The utility of the æther—developed in [6]—was first demonstrated in the building of large-scale intensional communities in author Mancilla’s Anita Conti Mapping Server [1]. Using the æther, the mapping server was able to connect the different components—graphical interface, geographical mapping tool, typesetter and multilingual database for geographical names—through broadcasting, without the need for explicit communication between the different components. The server allowed users to create maps according to personal settings, using a passive context, or allowed the creation of groups of users with one leader and followers, using the æther.

The æther proved useful, but due to its instantaneous response, changes could be passed in an inconsistent manner to the different applications, and the time for each application to respond appropriately might vary significantly, based on the tasks to be undertaken. With a large enough time lag between actions and with a small set of users, these problems were not significant. However, the solution did not scale, nor was it possible to model the direct communication that might take place between a group of users.
Specifically, if the delay between deltas to the æther were smaller than the delay in responding to the deltas broadcast from the æther, as could easily happen when building multiple-layered high-resolution maps on demand, then it would be impossible to guarantee anything about the aggregate state of the overall system. Each user would end up with an inconsistent state, when in fact the infrastructure was originally introduced precisely to share those aspects of the system that were to be held in common. For example, the created map for each user might well end up containing parameter values no longer valid in the æther.

The solution we propose here is that the interaction with the æther be restricted to follow a specific protocol, based on a discrete, synchronous model of execution. In each instant, there would be two stages. In the first, the æther would merge the deltas that it has received, and distribute the merged deltas to the applications. In the second, the applications would examine the new deltas, ensure that the changes they sent had been acknowledged, adapt to the deltas, continue with their processing, and in so doing ask for changes to be made to the common context. At the end of the instant, the accumulated changes would be wrapped up into a delta and sent to the æther for merging.

The proposed solution would greatly simplify the programming of many kinds of distributed systems, because the solution is deadlock-free. (It is not necessarily starvation-free, nor fair.) Writing to shared, global variables, which is what an æther encapsulates, is done in an atomic manner. If there is a clash during an instant, with two or more applications wishing to modify the value of the same dimension, then a merge operation would be applied over that dimension, with the possible results being a no-op, an error or the application of some sort of addition or averaging operator; in the next instant, the applications involved in this clash would be informed that a clash took place, along with the result of the clash resolution; subsequently, each application would decide how to handle this situation. Effectively, every write to the æther is guaranteed to be deterministic and atomic.

The paper is organised as follows. We give a brief summary (§2) of the basis for possible-worlds versioning, and how context-aware applications can be built using this intuition, and also how the æther can be used to share context between the applications. The core of the article is the presentation of the clocked æther (§3) and of the clocked context-aware program (§4), which lead naturally to the presentation of the synchronous community (§5). In the conclusions, we discuss the possibilities of implementation as well as the implications for interconnecting multiple synchronous communities.

2 Possible-worlds Versioning

Possible-worlds versioning—first proposed in [5]—and its history, is defined in detail in [2]. The reader is referred to that document for further details.

A possible world is a term from philosophy, dating back at least to John Duns Scotus (13th century C.E.), referring to a logically consistent world that is different in certain details from our own. During the twentieth century, possible-world semantics was used to give the semantics of utterances in natural language: the meaning of a sentence would be an intension: a mapping from possible worlds to ordinary values, called extensions.

In possible-worlds versioning, a possible world is a context, defined by a number of dimension-value pairs. In each possible world, there will be a complete copy of every entity: software, documentation, data bases, Web pages, build files, splash screens, etc. In a different world, there will also be a complete copy of every entity; for any given entity, it may be identical or slightly different to the one in the previous world, or it might be nonexistent. However, from an implementation point of view, there is maximal sharing, which succeeds through the use of a partial order defined over the universe of possible worlds.

We begin by presenting the context, the versioned object and the best-fit version.

**Definition 1** A context \( \kappa \in \mathcal{K} \) is a mapping

\[
\kappa := (d : v)^+ 
\]
where $d$ is a dimension and $v$ is a value.

To access the value associated with a dimension $d$, we write $\kappa(d) = v$. If $\kappa(d)$ is undefined, we write $\kappa(d) = \perp$. The empty context is written $\epsilon$. The domain of $\kappa$, written $\text{dom} \kappa$, consists of the set of dimensions for which $\kappa(d) \neq \perp$.

**Definition 2** Context $\kappa$ is less refined than context $\kappa'$, written $\kappa \subseteq \kappa'$, when $\forall d \in \text{dom} \kappa, \kappa(d) = \kappa'(d)$.

**Definition 3** Contexts $\kappa$ and $\kappa'$ are consistent if $\forall d \in \text{dom} \kappa \cap \text{dom} \kappa', \kappa(d) = \kappa'(d)$.

**Definition 4** The join of two consistent contexts $\kappa$ and $\kappa'$, written $\kappa + \kappa'$, is the union of $\kappa$ and $\kappa'$.

**Definition 5** Let $\mathcal{A}$ be a set of objects. A versioned object $\mathcal{A}$ of type $\mathcal{A}$ is a set of pairs $\mathcal{A} = \{ (\kappa_1, \alpha_1), \ldots, (\kappa_n, \alpha_n) \} \subseteq \mathcal{K} \times \mathcal{A}$.

The domain of $\mathcal{A}$, written $\text{dom} \mathcal{A}$, is the set $\{ \kappa_1, \ldots, \kappa_n \}$. If $(\kappa, \alpha) \in \mathcal{A}$, we write $\mathcal{A}(\kappa) = \alpha$.

**Definition 6** Let $\mathcal{A}$ be a versioned object and let $\kappa_{\text{req}}$ be a context. Then the best-fit version of $\mathcal{A}$ with respect to the requested context $\kappa_{\text{req}}$ is $(\kappa_{\text{best}}, \mathcal{A}(\kappa_{\text{best}}))$, where:

$$\kappa_{\text{best}} = \max \{ \kappa \in \text{dom} \mathcal{A} \mid \kappa \subseteq \kappa_{\text{req}} \}$$

The version tag is kept alongside the component in order to be able to compute the version tag of higher-level components, as seen below.

To change contexts, one uses context deltas. The versioned object is generalised to the delta-versioned object.

**Definition 7** A context delta $\delta \in \mathbb{D}$, or delta for short, is an operator for changing contexts:

$$\delta := \kappa \mid \overline{\nu} \quad (1)$$
$$\overline{\nu} := (d : \nu)^+ \quad (2)$$
$$\nu := \text{clear} \mid \text{set}(v) \quad (3)$$

When $\delta = \kappa$, it is absolute, and when $\delta = \overline{\nu}$, it is relative. The application of $\delta$ to $\kappa_0$, written $\kappa_0\delta$, is given by:

$$\delta = \kappa : \quad (\kappa_0\delta)(d) = \kappa(d)$$
$$\delta = \overline{\nu} : \quad (\kappa_0\delta)(d) = \begin{cases} v, & \overline{\nu}(d) = \text{set}(v) \\ \perp, & \overline{\nu}(d) = \text{clear} \\ \kappa_0(d), & \text{otherwise} \end{cases}$$

**Definition 8** Let $\mathcal{A}$ be a set of objects. A delta-versioned object $\overline{\mathcal{A}}$ of type $\mathcal{A}$ is a set of pairs $\overline{\mathcal{A}} = \{ (\delta_1, \alpha_1), \ldots, (\delta_n, \alpha_n) \} \subseteq \mathbb{D} \times \mathcal{A}$.

Delta-versioned objects are a generalisation of versioned objects. The domain of $\overline{\mathcal{A}}$, written $\text{dom} \overline{\mathcal{A}}$, is the set $\{ \delta_1, \ldots, \delta_n \}$. If $(\delta, \alpha) \in \overline{\mathcal{A}}$, we write $\overline{\mathcal{A}}(\delta) = \alpha$. Best-fitting with delta-versioned objects takes place with respect to a current context, by creating a versioned object first.

**Definition 9** Let $\overline{\mathcal{A}} = \{ (\delta_1, \alpha_1), \ldots, (\delta_n, \alpha_n) \}$ be a delta-versioned object and let $\kappa_{\text{cur}}$ be a context. The versioned object $\mathcal{A}$ generated from $\overline{\mathcal{A}}$ in the current context $\kappa_{\text{cur}}$ is given by: $\mathcal{A} = \{ (\kappa_{\text{cur}}\delta_1, \alpha_1), \ldots, (\kappa_{\text{cur}}\delta_n, \alpha_n) \}$.

The possible-worlds versioning article [2] presents the development of the æther by Paul Swo- boda and author Plaice. We do not present the existing æthers here, as we will be proposing a new form of æther in the next section.
3 The Clocked Æther

A clocked æther is a reactive program encapsulating a context with registered participants. At the beginning of instant $t$, an æther $\mathcal{A}$ contains:

- $\kappa_{t-1}$, the current context of the æther at the end of the previous instant;
- $P_{t-1}$, the current set of active participants at the end of the previous instant;
- $D_{p,t-1}$ (for each participant $p \in P_{t-1}$), the set of dimensions being listened to by participant $p$.

The æther $\mathcal{A}$ then receives, from each participant $p \in P_{t-1}$, a delta. In addition, a participant $p$ may change the set of dimensions to which it is connected. Suppose that the æther receives the following messages:

- $p^i_c$.$\text{connect}(D_i)$, $i = 1..m$
- $p^j_d$.$\text{delta}(\delta_j)$, $j = 1..n$

The new state of $\mathcal{A}$ is:

- $P_t = P_{t-1} \cup \{ p^i_c \mid D_i \neq \emptyset \} - \{ p^i_c \mid D_i = \emptyset \}$.
- $D_{p,t} = \begin{cases} D_{t}, & p^i_c$.connect$(D_i), \; i = 1..m \\ D_{p,t-1}, & \text{otherwise} \end{cases}$
- $\kappa_t = \kappa_{t-1} \delta_t$, where $\delta_t$ is defined below.

Computing the delta $\delta_t$ is more complicated, because different participants may have submitted incompatible deltas for the same dimension. Should this be the case for dimension $d$, then an associative and commutative operator $\bigoplus^d$ needs to be applied to merge all of the deltas for $d$.

The default $\bigoplus^d$ operator would produce an error value, designating that more than one writer attempted to write to the same dimension. However, other operators are envisageable: the no-op could mean that the writers would have to try again some other time, while the addition operator would be used to combine values from different writers.

The domain of $\delta_t$ is defined by:

- $\text{dom } \delta_t = \bigcup_{j=1..n} \text{dom } \delta_j$

For each $d \in \text{dom } \delta_t$, there are two possibilities:

- There exists exactly one $p^j_d$ such that $d \in \text{dom } \delta_j$. Then $\delta_t(d) = \delta_j(d)$.
- There exists $\{ j_1, \ldots, j_\ell \}$, $j > 1$, such that $\forall k \in 1..\ell, d \in \text{dom } \delta_{j_k}$. Then $\bigoplus^d$ is applied to the deltas: $\delta_t(d) = \bigoplus_{k=1..\ell}^d \delta_{j_k}(d)$.

Now that the new state of $\mathcal{A}$ has been computed, it is time to compute the deltas $\delta'_j$ to be sent back to the participants.

- $\delta'_j = \delta_t | D_{p,t} - \delta_j$

where $\delta_t | D_{p,t}$ is the restriction of $\delta_t$ to $D_{p,t}$. If there is no conflict for a dimension, there is no need to send back the delta for that dimension.

Therefore, the cycle of the clocked æther has two parts. First is the accepting stage, in which the æther accepts deltas from all the participant programs, combining the deltas into a single delta to be applied to its encapsulated context. Second is the broadcasting stage, in which the æther sends appropriate deltas to the participant programs.
4 The Clocked Context-aware Program

A clocked context-aware program $P$, designed to be used with $n$ clocked æthers $Æ_1, \ldots, Æ_n$, has the following structure in an instant:

$$
P ::= \text{in} ; B ; \text{out}
$$

$$
in ::= \delta_1.\text{receive} ; \ldots ; \delta_n.\text{receive}
$$

$$
out ::= \delta_n.\text{send} ; \ldots ; \delta_1.\text{send}
$$

where $B$ is a block of code. In other words, at the beginning of the instant, the program $p$ receives its updates from the æthers permeating it, executes its block of code, and then sends its own updates to the same æthers.

The block of code $B$ will need to examine the deltas that it has received, clear them, and then write to them, leaving these additional operations: $\delta.\text{read}(d)$, $\delta.\text{clear}$ and $\delta.\text{write}(d, E)$. Operations would also be needed for registering the monitoring of dimensions in the different æthers.

Because of the possibility of clashes when submitting deltas to an æther, $B$ would most likely begin with tests to determine whether all of the changes that were attempted actually took place. If they did not, then appropriate action would have to be undertaken, depending on the protocols of that particular community.

Should $B$ be written in a language supporting context-awareness, such as ISE [4], with context-dependent variables, function blocks and control flow, then $B$ would run with a current context $\kappa$. In the beginning of $B$, $\kappa$ would be adjusted to take into account deltas $\delta_1, \ldots, \delta_n$, including the management of the clashes.

Ideally, the language and libraries being used would allow a programmer to take advantage of the full possible-worlds versioning infrastructure described in §2, complete with delta-versioned objects. As the current context of an application changes, so would the structure of all the objects being manipulated.

Communication with other context-aware programs would take place by broadcasting via the æther, allowing every program to receive the information with a single action. Communication between a pair of programs would be done conceptually through a privileged pair of dimensions—one for each direction—even if the implementation would use a physical channel. This conceptual choice precludes instantaneous back-and-forth chatter between two programs.

The æthers that a program is connected to provide sets of global variables with control access, which ensures that all writes are guaranteed to be atomic, with appropriate feedback should they not be successful.

5 The synchronous community

A synchronous community consists of a clocked æther and a set of clocked context-aware programs connected to that clocked æther. The intuitions presented above are clear. Nevertheless, to make this proposal effective requires answering some key technical questions, most of which relate to access control over dimensions. Primitives need to be developed to define ownership, possibly partial, of dimensions; access (read, write, etc.) to these dimensions; clash-resolution operators ($\bigoplus^d$); etc. Developing these primitives will require building an implementation and doing experimentation.

Furthermore, the implementation can be made robust, in the sense that a community can continue to function normally if some of the programs drop out, by using timeouts and adding timestamp tags to each delta. Should, in a given instant, a program not send its delta in time, then the input delta for that program in that instant would be assumed to be null. Nevertheless, an output delta would be sent to that program, and the program could continue to remain synchronised with the æther. Several consecutive dropped deltas would make the æther assume that that program is no longer accessible and would no longer send it updates.
6 Conclusions

With the development of mobile and pervasive computing, the need for a framework in which to easily build communities of context-aware applications is clearly evident. The synchronous community, with its “breathing” rhythm of aethers working, then applications working, is a natural such framework.

The model of the synchronous community can be extended so that multiple communities can interact. Already, with this model, the different applications may register with multiple aethers, so long as the latter are “breathing” at the same rate, and that these aethers are not communicating among themselves. To extend the model would require mechanisms to ensure that deadlocks and livelocks do not take place when connecting multiple aethers.

References


