STICK-E NOTES: Change

Peter Brown
August 13, 1999

ABSTRACT

(PJB comment, Feb 2001: this was written in terms of Boolean rather than best-match retrieval. However, the concept of a unique ID still applies. The idea of using the previous context is more doubtful: if it was used at all, we would need to think in terms of the difference in score between a match of the current context with a match with the previous context.) Often the user of a context-aware application is primarily interested in change: information that applies to their newly entered context but not to their previous one, and new information that has arisen since they last contacted the system. This note looks at how triggering engines can make provision to show change.

Introduction

Here we consider various facets of change within our model of triggering. This model involves a triggering engine that has two sets of data:

(D1) some context-aware data, which we call *stick-e notes*.

(D2) the user's current context.

The triggering engine is called by an application, which specifies the data to be used and decides what to do with the resultant triggered notes, e.g. it might present certain fields to the user.

It is a basic assumption of context-awareness that the current context will be continually changing. It is, however, sometimes assumed that the stick-e notes are static, and indeed this is reasonable for many applications. In this paper we waive this assumption and explore the effect of constantly changing stick-e notes.

There are a few applications where the stick-e notes may change, but the current context is static (e.g. a user at a terminal keeping a watch on notes representing traffic problems), but such applications are not especially interesting. Thus we will assume that both the current context and the stick-e notes are changing.

Transitional information

One aspect of change that is important irrespective of what parts of the data are changing is the question of transitional information. An application that presents triggered information to the user can adopt one of two models:

(M1) it can call the triggering engine and display all the notes that are triggered.

(M2) it can call the triggering engine and display the notes that are newly triggered since the previous occasion. (It may also choose to point out the notes that were previously triggered, but no longer are: i.e. it might cover transitions both into and out of context.)

It is desirable that the triggering engine provide some aids to applications that wish to adopt approach (M2). Before considering this, we will first outline where the triggering engine gets its data from.

Where the data comes from

In general the current context may come from diverse sources: some of it may come from the application, some may come from *context servers* which communicate with the triggering engine, and some, as we shall see, may be supplied by the triggering engine itself. When we speak of the current context we mean the sum total of all of these.

In our current implementation of the triggering engine, stick-e notes are loaded (from whatever source) at the start of a session, and subsequently new notes may be added or deleted, perhaps through the external source sending messages to the triggering engine ('delete note about traffic jam X', 'add new note about jam Y', 'change description of jam Z'). This model offers efficiency advantages. However an alternative is to have all the data residing throughout in a database that is external to the triggering engine.

A way to provide transitional information

An approach to providing transitional information is to provide the triggering engine with *two* contexts, a *previous context* and the *current context*. It can then trigger those notes that match the current context but not the previous one. (Other combinations are also possible, e.g. the previous but not the current, though there may be logical problems if notes have been deleted from the system, as distinct from being marked as timed out, since the previous context was used.) Clearly this facility helps an application to provide option (M2); it also allows (M1) to be provided: the previous context can be specified as 'void', thus causing everything that matches the current context to trigger.

This mechanism is particularly easy to provide if the triggering engine has a single user, rather than acting as a server. In this case the triggering engine can remember the current context every time it does a triggering, and by default use this as the previous context next time. At the very start the previous context is treated as void.

When the triggering engine has many applications simultaneously using it, and in particular when it is a server on the web, the situation is more difficult. In our current implementation the triggering engine keeps no state information about the user, and thus each usage is treated as a completely new one. If it is desired to remember a previous context this is a task for the application. The application can then use the triggering engine later, passing back one of the previous contexts it has remembered (quite likely the most recent one). To help the application do this, the triggering engine provides, in its 'match report', the current context that was used.

Changes in notes and the use of `<uniqueID>` fields

The scheme as we have described it so far works well if the stick-e notes are static, but if we want to detect notes that are new since the previous context was used, we need to add a further facet: this is a `<uniqueID>` field added to each note. The uniqueID can be anything that increases with time, and thus can be used to detect if one note precedes another. The uniqueID could indeed be time itself, with the proviso that, if we really need the ID to be unique, no two notes should have the same time. In our model, where notes are loaded into the triggering engine, it can be a serial number, increased by one as each note is loaded. Assuming N1 is its current value, a loaded note will have the extra field

```
<uniqueID> N1 ..
```

added to it.

On the other side of the matching process, the user's current context is automatically enhanced by the extra field

```
<uniqueID> N2
```

where N2 is the current value of the uniqueID at the time the current context is supplied; thus it matches all the notes previously loaded. However if this same current context is used later as a previous context, it will only match the notes that were in existence at the original time. Given that the rule is to trigger notes that match the current context but not the previous context, the newer notes (i.e. those whose uniqueID matches the new current context but not the previous context) will be triggered, provided of course they match the rest of the new current context too. Thus the desired effect is achieved.

As the above implies, the `<uniqueID>` field is by default an active field — its initial score is zero — (but specifying `<uniqueID> none` within `<activeTags>` will override this). If a context already contains an explicit `<uniqueID>` field the automatic facility of adding it is suppressed. When automatically added the uniqueID field is an internal facility, hidden from the outside user: hence it does not appear in match-reports, and, if notes are only matched if one active field matches, a match of uniqueID does not count towards this.

Thus overall we have a mechanism that shows change in two ways: it both covers notes that have newly come into context and, given the uniqueID facility, notes that have been newly loaded. The mechanism only applies when the stick-e notes are information notes: in the case where the stick-e notes are rules, we have a naturally static situation, and the addition of extra uniqueID fields would be an unwelcome complication.

Granularity of change in a note

As we have described it, change is a property of a whole note. Sometimes this will lead to a triggering of a "changed" note which, as the user sees it, has not changed at all, thus causing puzzlement. For example a field of a note that in fact takes no part in the current matching process may change; if as a result the whole note is treated as a new one, it will be re-triggered as if there were something new to tell the user.

Thus there is a case for recording change at a lower level of granularity — at the field level for example. The downside is extra complexity.

Magnitude of changes

Sometimes it may be desired to trigger only when the value of a numerical field changes more than a certain amount, e.g. when a temperature field changes more than 5 degrees. The Matcher has no facilities to support this directly; the effect can, however, be achieved by making a sensor set a contextual field that records change, e.g. it might set two fields:

```
<temperature> 30
<change> 4
```

where the change relates to some time period determined by the application.

Time-out and note deletion

In the case where the triggering engine is a server running for large amounts of time, with new notes continually being loaded, there is a danger of it becoming clogged up. In some environments there is a case for a time-out facility whereby, after a certain time (which may be a global constant or a time attached to a particular note) each note is either:

- automatically deleted from the triggering engine's store.
- checked for validity against the original note source.

Time-outs are particularly apposite for 'correction' notes such as 'the traffic jam has now cleared'.

Time-outs can either be implemented within the triggering engine, as we have implied above, or by the original note source, e.g. a traffic reports service. In the latter case the service would send the triggering engine a message of form 'delete note X'.

Following on from this, there needs to be some way of specifying 'note X', such as a unique field attached to each note. This may be the same as the `<uniqueID>` field we have described above, or, given that the latter may be generated internally to the triggering engine, a separate field. A note that is loaded when the uniqueID is N1 and deleted when it is N2 can actually be left in the system, but given the uniqueID field

```
<uniqueID> N1 .. N2
```

Then it will not match later current contexts, but *will* match previous contexts whose uniqueID lies in the range N1 .. N2. This allows the Matcher to handle historical requests better, at the possible expense of holding material for ever.

Corrections for individual notes

Sometimes it is desirable to produce a new note that is only for people who have seen a certain previous note. A good example of this is 'this traffic problem has now cleared'. Clearly it would be tedious if users who had not seen the original message triggered this message.

To implement such a facility is is necessary to remember what notes the application has seen (this may be useful for logging anyway). Often this is best done on the application side. One way of doing this is for the triggering of a note to reset a field in the current context — specifically in that part of the current context store by the application. This is called the 'setting' facility. (The suggested 'glimpse' facility was similar, but was not fully explored.) In particular there can be a `<seenThis>` field in the current context which identifies all the notes that the application has seen. Notes can be indicated by their uniqueID. This field can then be used in subsequent matchings to achieve such facilities as:

- only showing a note once; this might be useful for correction notes anyway.
- only triggering one note if another has been previously seen.

Adding new fields

A further aspect of change is that, as time goes by, an application and its data formats will evolve, and new fields may be added. Perhaps new sensors will enhance the current context, and perhaps new fields will be added to existing notes. The need here is for existing applications, which may not be interested in these extra fields, to continue to work in the same way, although supplied with the extended data. In our model we try to achieve this by a careful choice of default behaviour, e.g. in the recipe for what is output and in the specification of which fields are active in matching.

Similar issues carry over to the problem of users with different hardware: e.g. user A is running an application on a PDA with no temperature sensor, but user B is running the same application on a PDA that does have a temperature sensor.

Conclusion

The property of change in context-aware systems has many facets, and applications that cater for it well are likely to make a variety of extra demands on a triggering engine. We have tried to show how the most important of these demands can be met. Two of the facilities are currently being added to our triggering engine: (a) the use of previous contexts to filter out old triggerings, and (b) `<uniqueID>` fields.