Building Components for a Distributed Sentient Framework with Python and CORBA

Diego López de Ipiña
Laboratory for Communications Engineering
Cambridge University Engineering Department
dl231@eng.cam.ac.uk

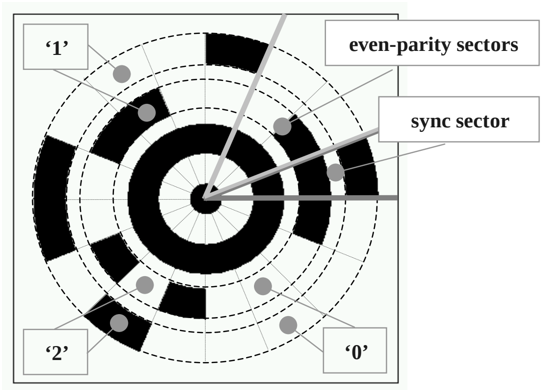
Abstract

TRIP (Target Recognition using Image Processing) is a novel vision-based sensor system that uses a combination of visual markers (2-D circular barcode tags, or *ringcodes*) and conventional video cameras to identify tagged objects in the field of view. A CORBA-based distributed component architecture called *Sentient Information Framework* has been devised to efficiently manage and distribute to applications the sensor data obtained both from TRIP and other sentient technologies. This paper describes the implementation, in Python, of two components for this framework; a *TRIP Directory Service* mapping ringcode identifiers to attributes and a *context abstractor* type component insulating low level details of sensor data acquisition and interpretation from an application. These case studies will reflect the potential of Python and CORBA, assisted by its *Event Notification Services*, as an ideal technology combination for the rapid development and efficient gluing of heterogeneous distributed software components.

1. Introduction

TRIP[1] (*Target Recognition using Image Processing*) is a new vision-based sensor technology that, by means of image processing and computer vision algorithms, processes video frames captured from cameras distributed through the environment to recognise 2D circular barcode (*ringcode*) tags attached to objects. The information inferred is the approximate location[2], orientation and identifier (*TRIPcode*) of the ringcodes (see Figure 1) spotted.

It is expected that this sensor technology will be useful in the realm of *Sentient Computing*, more commonly known as *context-aware computing* [13], a research field concerned with the ability of computing devices and applications to *detect*, *interpret* and *respond* to changing aspects of the user's context. Its goal is to enhance computer systems with a sense of the real world and make them know as much as the user about the aspects of the environment relevant to their application. To achieve this, it uses sensors distributed throughout the environment to maintain a detailed model of the real world and make it available to applications. Applications can then respond to environmental changes and autonomously change their functionality, without explicit user intervention, based on observations of *who* or *what* is around them, what they are doing, *where* they are and *when* something is happening.



* 10 **2011221210001**

Figure 1: *ringcode representing number 1160407.*

Location-aware computing, whose behaviour is determined by the position of objects in the environment, represents an interesting subset of the sentient computing paradigm since location is often an essential attribute of context. Conventional sensor technologies employed in the area of indoor location-aware computing involve a network of complex and expensive special purpose designed sensors to be deployed throughout an indoor environment, and the use of electronic battery-powered mobile positioning devices that transmit a signal via an infrared, ultrasound or radio wireless interface. AT&T's Active Badge [16] (infrared-based) and Active Bat [17] (ultrasound-based), and PinPoint Corp.'s 3D-iD [18] (radio frequency-based) location technologies are good

---

1 TRIP has been developed in collaboration with AT&T Laboratories, Cambridge

2 The current implementation only provides containment-based location i.e. it identifies the camera view range within which the *TRIP target* is sighted. Currently, the author is working on the improvement of the location resolution using stereo vision techniques.

examples of this. TRIP aims to demonstrate that similar usability levels can also be achieved, by using existing off-the-self technology (video cameras) to obtain the identity, orientation and location of passive cheaply printed ringcode tags (*TRIPtags*), with a less complex and more cost-effective infrastructure.

TRIP's current implementation, a C++ video filter [5] performing the target recognition process, processes 3 frames/second on a 400 MHz Pentium II machine and has accuracy levels in the range 98–100% when TRIPtags of 5x5 cm are viewed within three metres of the standard low-cost analogue cameras used. The special geometric features of the targets' design (see Figure 1) enable their recognition even when they occupy a very small number of pixels in the captured images and reduce the complexity of the computer vision task to a minimum.

A *TRIPtag* is a 2D black and white *ringcode* composed of:

- A central bull's-eye to make the identification process easier due to its invariance to rotation and perspective, and high contrast.

- A pair of encoding rings around the bull's-eye, divided into 16 sectors:
    - The 1st sector's (or *synchronisation sector*) encoding rings' combination, impossible anywhere else, indicates the beginning of a *TRIPcode*.
    - The 2nd and 3rd sectors implement a parity error checking mechanism.
    - The remaining 13 sectors represent a TRIP target ternary code in the range[3] 0 to 1,594,322. Observe a '0' is represented by leaving a sector's two code ring portions blank, a '1' by setting the sector's inner code ring portion to black and leaving the outer one blank and '2' is the reverse of this.

A *TRIP-aware Jukebox Controller* is the first application developed to demonstrate TRIP's context sensing capability. This application controls a virtual jukebox, implemented on top of an MP3 player, by using *TRIPtags* as a user interface device. A TRIPtag is used to identify a song, a person or jukebox control actions. When the TRIPtag representing a song is 'seen' by a camera attached to a *TRIP-enabled* computer, its automatic playback is initiated, either on that computer or on a nearby one with sound capabilities. Providing the TRIPtag worn by a person is spotted, the application produces the automatic selection of that person's play-list. Finally, some TRIPtags sightings trigger control actions on the jukebox (play, pause, etc.).

In the development of this application the following issues were raised:

1. A file for each TRIPcodes' category used (people, music-tracks and control-actions), containing a mapping between each TRIPcode and the attributes required to drive the final application had to be created. For example, the identifier representing a song had to be mapped to the file path of such track in the MP3 archive.

2. The set of TRIPtags used to control the application had to be manually generated by providing the label, identifier and size attributes of each tag to a POSTSCRIPT generating script.

3. The final application logic was complicated due to the need to:
    a) Filter the raw contextual data provided by TRIP to determine if the TRIPcode sighted really corresponded to any of the TRIPcodes of interest.
    b) Transform this data into information directly usable by the MP3 player (e.g. the file location of a song).

These three issues led to the development of the *Sentient Information Framework* (SIF), a programming framework that streamlines and facilitates sentient application development by isolating context capture and abstraction from application semantics and providing efficient mechanisms for context communication. This work is inspired by previous research efforts made with the same aim of managing and disseminating contextual information efficiently, such as the SPIRIT [4], Context Architecture [2] or Context Information Service [12] projects. The rest of this paper describes the SIF framework, focusing its attention on analysing the benefits of having adopted Python for the implementation of some key components of it.

2. The Sentient Information Framework

The *Sentient Information Framework* (SIF) architecture consists of a group of co-operating distributed software components that use events as a uniform way of informing each other of context notifications. *Context Channel* objects that are both suppliers and consumers of sentient data are interleaved in-between SIF components, enabling multiple suppliers to transparently and asynchronously communicate with

---
[3] Note $3^{13} = 1{,}594{,}323 \approx 2^{20}$ valid codes

multiple consumers without the components knowing about each other.

OMG's CORBA [9] is the chosen middleware because of its platform and language inter-operability features, its popularity as an open standard for component software operation and its rich set of standard distributed services. The *CORBA Event Service* [10] is used to enable SIF components to *asynchronously* notify contextual events, to interested applications or other SIF components, through Event Service's Event Channels (renamed as *Context Channels* in SIF). If the Event Service had not been used, each SIF component would have had to perform a *distributed callback* for each event notification for all the clients registered. In addition, this service adoption permits interested parties to establish communication with SIF components, following either a *push* or *pull* event communication model. With a *push* model, the event producer takes the initiative and pushes events to an *Event Channel* that then subsequently sends them to consumers. For the *pull* model, the consumer issues a pull request on the Event Channel object and this, in turn, pulls the event data from the supplier. This event pull request can be blocking (`pull` method) or non-blocking (`try_pull` method).

New components can be integrated into SIF as *context consumers*, *context suppliers* or as both simultaneously. The only requirement for a component to form part of SIF is to be CORBA-enabled and to adopt the OMG Event Service for contextual event communication, regardless of the platform and programming language used. Consumer components register with Context Channels that serve their context notifications of interest. Supplier components either create new Context Channels or reuse previously created ones by other components generating the same event types, where, after registering, communicate their own contextual events. In order for the different SIF components to know about each other, every component binds its object reference with the CORBA *Naming Service* under a humanly recognisable name. Components interested in other component services contact the Naming Service, providing well-known component names, to obtain their CORBA object references. The following three component classes are found in SIF (see Figure 2):

1. *Context Generators (CG)*
   These encapsulate a single sensor or a set of related sensors and the software that acquires raw context information from them. The information obtained is transferred to *Context Channels* in event form, following usually a *push* event communication model. CGs could be based on TRIP or other sensors such as Active Badge [16], GPS, or a simple microphone.

2. *Context Abstractors (CA)*
   Seen by applications as *proxy context generators*, they achieve the separation of concerns between context sensing and application semantics. They consume the raw sentient data provided by CGs, interpret its contents and augment them with static data from a database, producing enhanced contextual events that can directly drive final applications. Sometimes CAs also *correlate* other CAs' or CGs' outcomes to generate the demanded sentient data.

   These components' operation responds to an Event-Condition-Action (ECA) model. They monitor for asynchronous event communication, apply conditional statements over them and whenever a condition is fulfilled, generate an action (in this case, produce a higher level event). Between a CG and a final application, an event can flow through N-tiers of CA *active* components. For example, a person's TRIPtag sighting event, "code 1223 seen by camera 30", could be transformed by a *location context abstractor* into "Diego seen in room 1's camera 3's view range". In turn, this last event could be fed into a *lab access context abstractor* that would convey an "open door" event providing a TRIPtag wearer stands in front of the lab main door. Alternatively, a *teleporting context abstractor* receiving the same event from the *location* CA would, for example, issue a "move Diego's desktop to computer 5" event when the user is sighted close to computer 5 location. The abstracted events generated would finally reach the sentient applications in charge of performing the actual action, i.e. an *access monitor* and a *teleport* application.

3. *Context Channels (CC)*
   These are the intermediary entities that *de-couple* the communication among components of the previous two types and final applications. They constitute the *glue* that enables the *heterogeneous* software components and applications conforming to this architecture to inter-operate and are physically implemented as OMG Event Channels. They are shared by co-operating components that either generate or consume the same class of events, providing a means to classify and separate sentient events.

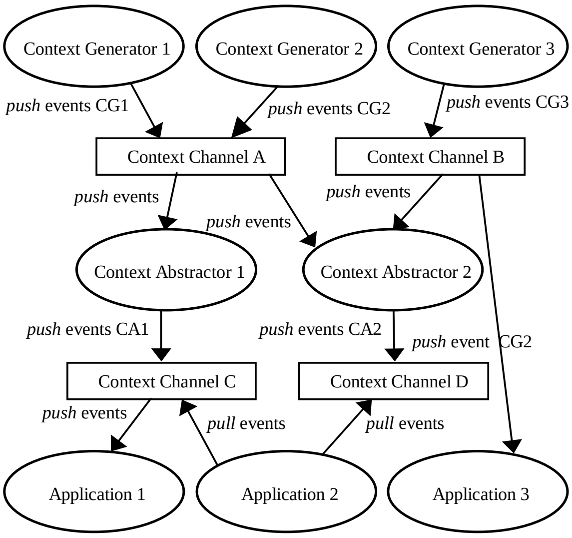


Figure 2: *The SIF Architecture*

OMG Event Service's suppliers, consumers and Event Channels handle event data in the form of the IDL[4] (Interface Definition Language) `Any` type[5], which enables components to send and receive domain-specific event data without requiring Event Channels to understand event data types. This implies that every consumer must know what type to expect in the `Any` they receive. Event Channels are only concerned with event distribution and, therefore, they require a filtering process in the event receivers to eliminate undesirable events, unfortunately leading to an increase in the required network bandwidth and consumers' processing load, as has been well discussed in [14]. The OMG Notification Service [11] addresses the limitations of the Event Service and supplies not only event filtering but also control over the quality of service that an Event Channel (here *Notification Channel*) provides. Future work on SIF will adopt this facility.

3. The TRIP Monitoring Service CG

The key component of the TRIP technology is a C++ written video filter software [5] that performs the TRIP target recognition process over video frames supplied by a set of networked cameras. In order to make the output of this environment monitoring process available to interested parties, the TRIP video filter has been encapsulated in a distributed CORBA component named *TRIP Monitoring Service*. Its operation is de-coupled from its consumers via an interleaved *TRIP Monitor Context Channel* where it *push*es TRIP sighting events. Figure 3 shows the IDL code defining a TRIP sighting event structure and the interfaces offered to clients to connect to its CC as either *push* or *pull* event consumers[6].

4. The TRIP Directory Service

The *TRIP Directory Service* component aims to provide a centralised body that regulates the TRIPcode granting process, stores static properties associated with TRIPcodes, and provides interfaces for their query. This Python written distributed component performs the following operations:

1. Creation, modification and deletion of new TRIPcodes categories.
2. Creation, modification and deletion of TRIPcodes and attributes associated with them.
3. Retrieval of categories' TRIPcodes and subcategories details.
4. Retrieval of a given TRIPcode's details.

It is assumed that clients of this component, for efficiency purposes, will on initialisation retrieve the properties associated with their TRIPcodes' subset of interest. Occasionally, clients will be long-lived and in

```
module TRIPMonitor{
  interface TRIP {
    // Event interface:
    CosEventChannelAdmin::ProxyPushSupplier getPushEventSupplier();
    CosEventChannelAdmin::ProxyPullSupplier getPullEventSupplier();

    // Event structures:
    struct TRIPevent {
      string TRIPcode; // code ternary representation
      paramsEllipse params; // bull's-eye outer ellipse parameters (x,y,a,b,θ)
      string cameraID; // capturing camera identifier
    };
    (…)
  };
};
```

Figure 3: *IDL interfaces for the TRIP Monitoring Service*

[4] IDL is a descriptive language that supports C++ syntax for constant, type, and operation declarations and lets one specify components' boundaries and their interfaces with potential clients.

[5] The `Any` type is a CORBA IDL built-in type that can represent any possible IDL data type, whether it is built-in or user-defined.

[6] These two interfaces must be provided by every SIS component that communicates contextual events to a Context Channel.

the mean time their information of interest will change in the Directory Server. Thus, this component requires an *asynchronous notification mechanism* to notify to its clients in real-time when the TRIPcode categories they use are modified. This facility converts this component into a *context generator*.

4.1. Reasons for Implementing in Python

Python is a general-purpose dynamic language that brings the power and flexibility of scripting to large software systems, like the one in mind. Moreover, it satisfies the three main programming requisites of this component: (1) a key-based (TRIPcode) object persistence mechanism through its `shelve` module, (2) CORBA support thanks to the existing Python CORBA mappings and (3) an asynchronous notification mechanism by means of the CORBA Event Service. Alternatively, a system-level programming language such as C++ or Java could have been adopted to provide higher performance, especially considering the fact that for the *TRIP Monitoring Service*, CORBA and C++ had already been used. However, in our case Python's speed of development determined our choice despite the obvious sacrifice of execution efficiency.

Python's `shelve` module provides a key-based persistence mechanism that enables the straightforward creation of persistent associative arrays of objects. Creating or accessing `shelve`'s instances is as easy as manipulating items in a normal Python dictionary. Keys in a *shelf* are ordinary strings but the values can be arbitrary Python objects, anything that the `pickle`[7] module can serialise into character streams. `Shelves` facilitate the creation of databases of native Python objects because there is no need to deal with another API, manage database-specific record structures, or convert to and from them when interfacing with the database. From C++ the UNIX *dbm* [3] library could have been used instead. However, *dbm* files only allow string-to-string associations and object serialisation in C++ must be hand coded. Java, on the contrary, provides good object serialisation features but lacks support for serialising objects by key through a convenient hash-file mechanism. The emerging JNDI (Java Naming and Directory Interface) [7] aims to overcome this Java deficiency but provides a much more complicated API than `shelves` because it is conceived as a universal interface to existing commercial Directory Services.

[7] *shelve* mechanism's performance is very acceptable when the C implemented version of pickle, cPickle, is used.

The existing OMG IDL to Python mappings supported by either Xerox's ILU [6] or University of Queensland's Fnorb [1] products provide Python programmers with CORBA distributed systems programming capability. The simplicity of the mapping, compared to C++ and Java, makes these CORBA implementations ideally suited as a tool for the rapid prototyping and scripting of CORBA systems and architectures. Python's object-oriented features integrate seamlessly with the CORBA Object Model. Python releases programmers from the complicated memory management issues of the CORBA C++ mapping. The language's dynamic features remove the need for tedious type-casting and long variable declarations as occur in the C++ and Java mappings for CORBA. Python's high code density is further emphasised when dealing with CORBA programming. CORBA eases distributed systems programming but requires a considerable amount of extra code to be added to programs – Python minimises this. Fnorb was the Python CORBA mapping chosen because, unlike ILU, is Python and CORBA/IDL specific, which makes it simple, lightweight, and easy to install and use. It supports all CORBA 2.0 data types (including `Any`) and provides a full implementation of IIOP[8].

CORBA's clear separation of implementation and interfaces, by means of IDL, makes possible to re-implement a given component in a more efficient programming language without having to modify the code of its clients. If scalability or performance problems appear in the future, the TRIP Directory Server could be re-coded in a better performing programming language and/or a commercial DBMS engine or X.500/LDAP Directory Service could be used instead of `shelve`. Furthermore, CORBA adoption opens the TRIP Directory Service to clients implemented in different programming languages and running on distinct platforms and enables this Python component to benefit from the existing rich set of standard CORBA services such as the Naming Service and Event Service.

4.2. Server Implementation Issues

4.2.1. TRIP Directory Server Persistent Dictionaries

The TRIP Directory Service is logically formed by a tree-based structure containing category nodes' sub-trees beginning from an initial category node named *root*. The children of a category node are either sub-category nodes or TRIPcode nodes; TRIPcode nodes

[8] Internet Inter-ORB protocol that allows the communication between different vendors ORBs, such as in our case when we communicate between omniORB2 and Fnorb ORBs.

are always leaves in the tree. Physically the TRIP Directory Service's tree is materialised into the following two Python persistent dictionaries:

1. The *Categories Shelf* contains category nodes hashed by `categoryKey`. A `categoryKey` is a string with the format $(xxx)^{+}$, where *xxx* is a three-digit ternary code in the range[9] 000 to 212 and '+' denotes one or more of these sequences. Each category node is, at the same time, a dictionary by itself. Figure 4 shows the contents of this dictionary. The `categoryID` key maps to a category identifier string in the form $root(.subCategoryName)^{*}$, where * stands for zero or more times. The `subcategories` and `TRIPcodes` keys point to counters indicating the number of subcategories and TRIPcodes in a category, respectively. `deleted_subcategories` and `deleted_items` keys hash to a list of deleted subcategories' keys and a list of deleted TRIPcodes, whose addresses can later be reused.

```
categoryKey  ──────▶  { categoryID            : root(.nameSubCategory)*
(e.g. 000122)           subcategories         : #
                        TRIPcodes             : #
                        deleted_subcategories : [categoryKeys]
                        deleted items         : [TRIPcodes] }
```

Figure 4: *Categories Dictionary Node*

```
TRIPcode ──────▶  TRIPcodeDetails(label, propertiesList), where:
(e.g. 0001222200101)  propertiesList = [Property(propertyName, propertyValue)]
```

```
module TRIPDirectoryService {
  struct Property {
    string propertyName;
    any    propertyValue;
  };
  typedef sequence<Property> PropertyList;

  struct TRIPcodeDetails {
    string label;
    PropertyList propertiesList;
  };
  (…)
};
```

Figure 5: *TRIPcodes Dictionary Node and its associated IDL structures*

```
module TRIPDirectoryService {
  interface TRIPDirectoryServiceIF {
    // Categories Dictionary manipulation interfaces
    boolean createCategory(in string parentCategoryID, in string categoryName);
    (…)
    // TRIPcode Dictionary manipulation interfaces
    string grantTRIPCode(in string categoryID);
    void   saveTRIPcode(in string TRIPcode, in TRIPcodeDetails data);
    (…)
    // Query interfaces for Categories Dictionary
    stringList getSubCategoriesList(in string categoryID);
    (…)
    // Query interfaces for TRIPcodes Dictionary
    TRIPcodeDetails getTRIPcodeDetails(in string TRIPcode);
    (…)

    // Event Interfaces
    CosEventChannelAdmin::ProxyPushSupplier getPushEventSupplier();
    CosEventChannelAdmin::ProxyPullSupplier getPullEventSupplier();

    // Event structures:
    struct AddTRIPcodeEvent {
      string          categoryID;
      string          TRIPcode;
      TRIPcodeDetails details;
    };
    (…)
  };
};
```

Figure 6: *TRIP Directory Server IDL interfaces*

[9] Note the codes 220, 221 and 222 are reserved because the prefix '22' denotes the beginning of a valid TRIPcode.

2. The *TRIPcodes Shelf* associates a `TRIPcode`, the ternary representation of a number in the range 0 to 1594322 (2^{13}-1), to a Python mapped CORBA IDL structure containing as members a sequence of name/value pairs and a `label` (see Figure 5). Note IDL structures are mapped into Python classes with a public attribute for each member of the structure.

For each category up to 24 subcategories can be created (range of ternary codes 000 to 212). When a new subcategory is assigned, the identifier of the new category is formed by adding to the parent's `categoryKey` the following non-used ternary code string in the mentioned range. A `TRIPcode` is composed of a prefix with the key of its category, followed by the ternary string '`22`', and the remaining ternary digits up to 13, the design of TRIP targets address supports, with the representation of the TRIP target sequence number within its category.

4.2.2. TRIP Directory Server Functionality

Figure 6 lists some of the IDL interfaces provided by the TRIP Directory Server to enable clients to manipulate and query its contents. It also shows the event interfaces that permit clients to register as event consumers of the *TRIP Directory Server Context Channel*, where category modifications are notified. The OMG Event Service's implementation *omniEvents*, provided by AT&T's C++ ORB omniORB2 [8], was used to provide this server's asynchronous communication mechanism. On initialisation, the server obtains from the CORBA Naming Service (*omniNames* of omniORB2) an object reference to a registered *Event Channel Factory* object. It then invokes the method `create_object` in this factory to generate the *TRIP Directory Server Context Channel* and connects to it as a *push* event supplier. Figure 7 shows how this registration process is implemented in Python. It might prove a little difficult to understand without a thorough study of the OMG Event Service [10]. When the TRIP Directory Server's shelves are modified, a notification is *pushed* to the previously obtained channel indicating the type and attributes of the modification. Figure 8 shows the Python code for sending a TRIPcode creation notification (`AddTRIPcodeEvent` IDL structure in Figure 6). Once an event is received at the CC, this takes the responsibility of delivering it to all registered consumers.

5. A GUI-based front-end for the TRIP Directory Server

In order to provide a user-friendly way to manage the query, creation, deletion and manipulation of TRIPcodes and categories, and the TRIPtag generation, a GUI front-end client for the TRIP Directory Server has been created.

5.1. Reasons for Implementing in Python

Once again a decision had to be made regarding choice of programming language. We discarded C++ because we wanted a Directory Service front-end that would run on all of our platforms (Windows NT 4.0 and RedHat

```
# Get a reference to the initial naming service context
initialContext = orb.resolve_initial_references("NameService")
# Create name under which EventChannelFactory is bound with NamingService
name = [CosNaming.NameComponent('EventChannelFactory','EventChannelFactory')]
(…)
# Lookup the EventChannelFactory in the naming service
obj = initialContext.resolve(name)
# The Naming Service object references returned is downcasted to the proper type
eventChannelFactory = obj._narrow(CosLifeCycle.GenericFactory)
(…)
#  Obtain an Event Channel instance from the EventChannelFactory
eventChannel = eventChannelFactory.create_object(…);
(…)
# Obtain a reference to the Event Channel's Factory of Proxy Push Consumers
supplierAdmin = self.eventChannel.for_suppliers()
(…)
# Create an instance of the Event Channel's proxyPushConsumer
self.consumer = supplierAdmin.obtain_push_consumer();
(…)
# The TRIP Directory Server connects itself as push Event Supplier
self.consumer.connect_push_supplier(self)
```

Figure 7: *TRIP Directory Server registration to its Context Channel as Push Event Supplier*

```
(…)
typeCode = CORBA.typecode(CORBA.id(TRIPDirectoryService.AddTRIPcodeEvent))
data     = TRIPDirectoryService.AddTRIPcodeEvent(categoryID, TRIPcode, detailsTRIPcode)
self.consumer.push(CORBA.Any(typeCode, data))
(…)
```

Figure 8: *`AddTRIPcodeEvent` transmission to Context Channel*

Linux 6.1) without having to recompile our code or use different GUI toolkit libraries. Java seemed to be a good candidate due to its multi-platform portability, CORBA support and excellent GUI toolkits (AWT or Swing). However, Python was chosen because it provides these same facilities and it has GUI toolkit libraries that are even easier to use than these of Java, requiring fewer code lines to achieve GUIs of similar sophistication. The main interest was to rapidly prototype this GUI-based client, without any special real-time execution performance requirements, making Python undoubtedly the best choice. Pmw [15], a toolkit for building high-level compound widgets in Python using the Tkinter module, was used because of its rich set of widgets and its multi-platform portability features.

5.2. Implementation Issues

The TRIP Directory Server GUI-based client (see Figure 9) is divided into two main interaction panes. The *TRIPcode Manager* Pane permits the user to: (1) browse through the existing TRIPcode categories displaying their subcategories and TRIPcodes, (2) create, modify and delete subcategories, and (3) create

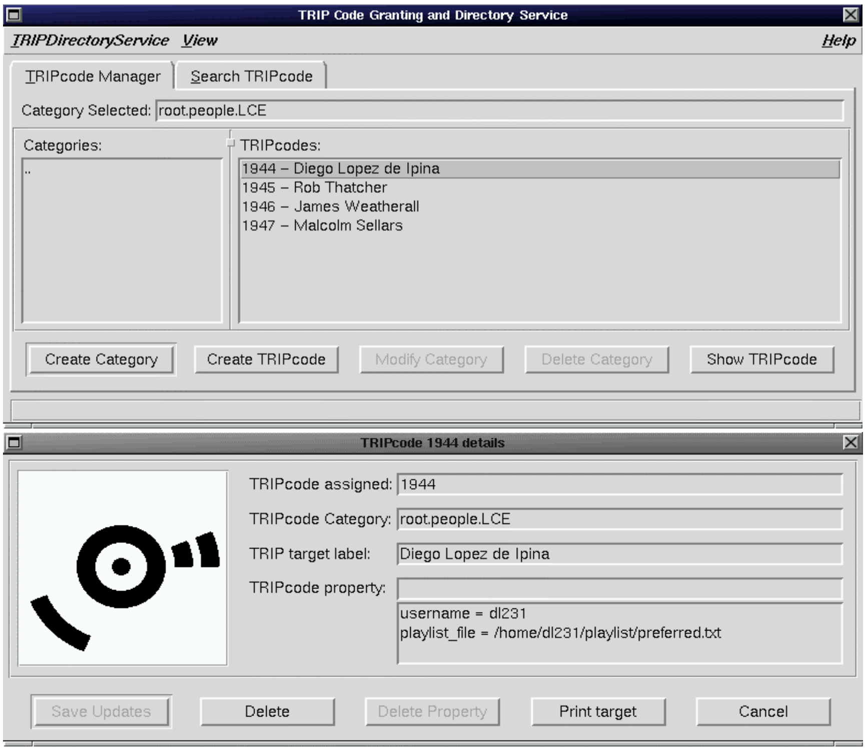


Figure 9: *Snapshots of the TRIP Directory Client*

```
def showTRIPcode(self):
  # Obtain the selected item from the TRIPcodes list
  TRIPcodeSelected = self.TRIPcodeList.component('listbox').get(Tkinter.ACTIVE)
  # Extract the TRIPcode identifier from the selected item string
  index = string.find(TRIPcodeSelected,' ')
  TRIPcode = TRIPcodeSelected[:index]

  # Invoke a RPC in the TRIPDirectoryServer to obtain the details of the TRIPcode
  TRIPcodeDetails = self.TRIPDirectoryServer.getTRIPcodeDetails(ternary(TRIPcode))
  categoryID = self.categoryEntry.get()

  # Create a modal dialog visualising the contents of the TRIPcode selected
  TRIPcodeVisuDialog(TRIPcode, categoryID, TRIPcodeDetails)
```

```
def getTRIPcodeDetails(self, TRIPcode):
  self.__lk.acquire() # Lock access to TRIPcodesShelve for the server thread executing this method
  if self.TRIPcodesShelve.has_key(TRIPcode):
    TRIPcodeDetails = self.TRIPcodesShelve[TRIPcode]
  else:
    TRIPcodeDetails = TRIPDirectoryService.TRIPcodeDetails('', [])
  self.__lk.release()
```

Figure 10: *TRIP Directory Client and Server method invocation after double click over TRIPcode list item.*

TRIPcodes within a category. On the other hand, the *Search TRIPcode* pane provides the means to (1) query the information associated with a given TRIPcode, (2), add, modify and delete its properties and (3) print a TRIPtag. Figure 9 shows the result of double clicking over a TRIPcode list item in the *TRIPcode Manager* Pane. Figure 10 illustrates in its first part the GUI *callback* method invoked when such action is performed and in the second the server implementation of the method `getTRIPcodeDetails` remotely called by the client.

6. Implementing a CA in Python

Context abstractor components aim to translate incoming raw sensor data (e.g. *TRIP target 2345 spotted*) into augmented contextual data directly usable for applications (e.g. *play song event*). The further layers of abstraction they provide insulate applications' logic from sensor data gathering and interpretation. The Python implementation of a *TRIP-aware context abstractor* for the *Jukebox Controller* application, described in section 1, illustrates the general procedure to be followed when implementing other components of this type. Its mission is to provide the virtual jukebox with context notifications that it can directly understand. To achieve this, it (1) filters out target sightings that do not correspond to the domain of the application and (2) interprets raw valid target sighting events generating the actual event types required to drive its operation.

Potentially the virtual jukebox could also be controlled by the signals generated from another context generator, e.g. an infrared remote control. A *remote control context abstractor* would gather the infrared-modulated code signals received by an infrared sensor CG, interpret them and *push* jukebox control events to a shared context channel with the TRIP-aware context abstractor. In this way, the final application would transparently respond to the control events received, regardless of their origin. Python is ideal for the implementation of these CAs because of its rapid prototyping capability and CORBA support. These components are usually not very computation-intensive, they just need to process and/or combine events obtained from CCs, producing as outcome enhanced sentient notifications.

Figure 11 represents the flow of interaction among some SIF components and the *Jukebox Controller* application. When the *TRIP-aware Jukebox controller context abstractor* is started, it obtains from omniORB2's Naming Service object references for the two *heterogeneous* CORBA components with which it will interact: the Python implemented *TRIP Directory Server*, over Fnorb ORB, and the C++ implemented *TRIP Monitoring Service*, over omniORB2. The obtained TRIP Directory Server reference is used to retrieve the details associated with the three TRIPcodes' categories of interest for the context abstractor:

1. *root.people.LCE*: TRIPcodes associated with LCE lab members.

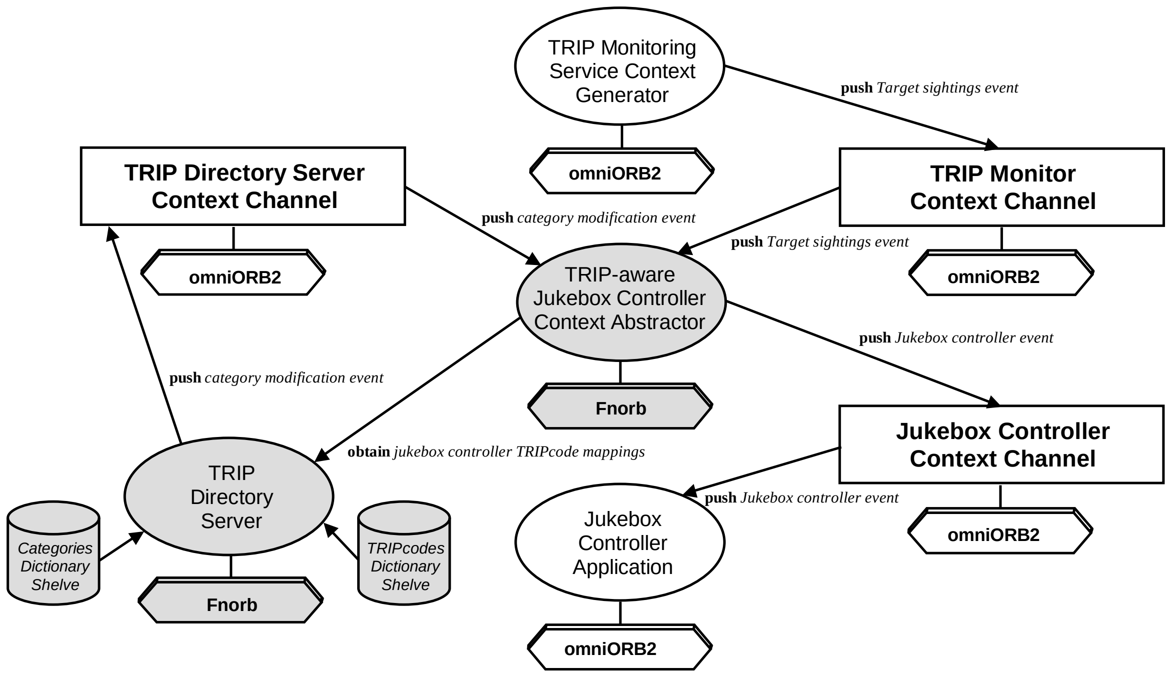


Figure 11: *TRIP-aware Python (grey) and C++ (white) SIF components and application*

2. *root.object.music-tracks*: TRIPcodes representing MP3 tracks to be played.
3. *root.action.jukebox*: TRIPcodes depicting jukebox control operations (*play*, *pause*, etc.)

Next, the Jukebox abstractor has to register as *push event consumer* of both the *TRIP Monitoring* and *TRIP Directory Servers'* Context Channels. For this, the abstractor must:

1. Implement the OMG Event Service `CosEventCom::PushConsumer` [10] interface (see Figure 12).
2. Invoke the method `getPushEventSupplier` in both servers to obtain *proxy push supplier* object references of their context channels.
3. Connect to the proxy push event suppliers.

Figure 13 shows stages 2 and 3's implementation. Note the TRIP-aware Jukebox Controller CA will receive event notifications from two different context channels through the same *distributed callback* interface (`push`).

Finally, the context abstractor looks up in the Naming Service for a registered *Jukebox Controller Context Channel* and, if one exists, obtains its object reference. (If one does not exist, then it creates a context channel instance from an Event Channel Factory, found through the Naming Service, and binds the obtained channel reference with the Naming Service). Either way, then, it registers as a *push event supplier* of this CC, using code similar to that in Figure 7. The IDL code of the events this CA conveys to its channel and the interfaces provided for clients to connect to it are shown in Figure 14. Whenever an event from the TRIP Monitoring Service or the TRIP Directory Service is received, the context abstractor's `push` method is invoked. Providing the notification originates at the TRIP Monitoring Service, and if it corresponds to a jukebox related TRIPcode, then after its processing, one of the jukebox control events of Figure 14 is pushed. Otherwise, the event received, coming from the TRIP Directory Server, is checked to determine whether the category modification has been done in any of the jukebox related categories, and if it is so, it updates its in-memory TRIPcode details. Figure 15 illustrates the `push` method's implementation.

7. Conclusion

This work has shown the ample range of capabilities offered by the integration of Python and CORBA, demonstrating Python's space in the development of CORBA distributed software components. Python has served us to develop a full-fledged Directory Service for our novel sensor technology that if implemented with a system-level programming language, such as

```
module CosEventComm {
  interface PushConsumer {
    void push (in any data) raises(Disconnected);
    void disconnect_push_consumer();
  };
  (…)
};
```

Figure 12: *OMG Event Service* `PushConsumer` *Interface*

```
supplierTRIP = self.TRIPMonitorServer.getPushEventSupplier()
supplierTRIP.connect_push_consumer(self)
supplierDirectory = self.TRIPDirectoryServer.getPushEventSupplier()
supplierDirectory.connect_push_consumer(self)
```

Figure 13: *Context Abstractor registration to Context Channels*

```
module JukeboxAbstractor{
  interface JukeboxAbstractorIF {
    // Event interface:
    CosEventChannelAdmin::ProxyPushSupplier getPushEventSupplier();
    CosEventChannelAdmin::ProxyPullSupplier getPullEventSupplier();

    // Event structures:
    struct playlistEvent {
      string playlistFilePath;
    };
    struct mpg3TrackEvent {
      string songToPlay;
    };
    struct actionEvent {
      string action;
    };
    (…)
  };
};
```

Figure 14: *Event interfaces for Jukebox Context Abstractor*

```
def push(self, event):
  eventType = event.typecode().name()
  eventData = event.value()

  if eventType == "TRIPevent":
    # Process the events received from TRIP Monitor Context Channel
    if string.find(eventData.TRIPcode, self.lceCategPrefix) == 0:
      # TRIPcode represents a member of LCE
      propertiesList = self.people[eventData.TRIPcode].propertiesList
      index = 0
      while (propertiesList[index].propertyName != "playlist_file") and
            (index < len(propertiesList)):
        index = index + 1
      if index != len(propertiesList):
        # Push an event of type playlist to the Event Channel
        typeCode = CORBA.typecode(CORBA.id(JukeboxAbstractor.playlistEvent))
        data = JukeboxAbstractor.playlistEvent(propertiesList[index].propertyValue)
        self.consumer.push(CORBA.Any(typeCode, data))
    elif string.find(eventData.code, self.trackCategPrefix) == 0:
      # TRIPcode represents an MP3 song track
      (…)
    elif string.find(eventData.code, self.jukeboxActionCategPrefix) == 0:
      # TRIPcode represents an jukebox action
      (…)
    else: # Filter out the event
      pass
  # Process the events received from TRIP Directory Server Context Channel
  elif eventType == "AddTRIPcodeEvent":
    # Determine category in which new TRIPcode was added
    if eventData.categoryID == "root.people.LCE":
      # Update people dictionary with the new TRIPcode of a person
      self.people[eventData.TRIPcode] = eventData.details
    elif eventData.categoryID == "root.action.jukebox":
      # Update jukebox operations dictionary
      (…)
    elif eventData.categoryID == "root.object.music-tracks":
      # Update MP3 tracks dictionary
      (…)
    else: # Ignore modification in categories of no interest
      pass
  elif eventType == "DeleteTRIPcodeEvent":
  (…)
```

Figure 15: *Jukebox Controller Context Abstractor* `push` *method implementation*

C++ or Java, would have required a much longer development time than the three programmer-weeks it took. Python has also assisted us in the rapid development of *context abstractor* type components that enabled us to experiment with, and show the potential of our sensor technology (TRIP) and the SIF architecture. The Jukebox Controller application has been successfully re-coded and integrated with SIF.

Further context abstractors and applications will be developed in Python, to explore new application domains for TRIP. For example, a planned context abstractor will store TRIPtag sightings indexed by location and timestamp to permit context-based retrieval applications. In addition, new context generators will be created to combine TRIP's sentient data with inputs from other sensors. This process will lead to the generation of a catalogue of reusable and extensible SIF components. The CORBA Trader Service [10], which defines a yellow pages service classifying CORBA object references by object properties, will be useful in its creation.

This work reveals that Python's known extensibility and gluing capabilities with higher performance languages, such as C, C++ or Java, can be further increased thanks to integration with CORBA and the usage of its event notification services. The case study described in Section 6 illustrates how OMG Event Channels can be used as *glue* for heterogeneous distributed software components.

The SIF architecture requires several improvements, the most critical one being the replacement of the Event Service by the Notification Service [11]. This modification will release context abstractors from the event-filtering stage and reduce event transmission bandwidth. Event consumers will specify at their registration the conditions or constraints the events they wish to receive must satisfy, and the Notification Channels will carry out the filtering process for them.

Never has the development of CORBA distributed applications been such a simple and fast process as it is with Python. Very little CORBA literacy is required from the programmer to produce working distributed applications. Memory management is done automatically by Python and, due to its dynamic properties, inconvenient long CORBA variable name declarations are eliminated. The Python programming community has much to gain from the existing CORBA Python mapping implementations, providing they become a little more robust and faster. The newly appeared omniORB2's CORBA Python binding (*omniORBpy*) promises much on this aspect. Its performance and robustness will be explored in future work. Although, due to Python's performance constraints, mostly client-side CORBA systems will be developed in this language, still they can benefit much from both the ample set of existing standard CORBA services and Python's excellent standard library for the development of sophisticated distributed systems.

Acknowledgements

The author is very grateful to Sai-Lai Lo for his expert advice and help on the TRIP project and to Frank Stajano for introducing him to Python and encouraging him to write this paper. He would also like to thank AT&T for the industrial sponsorship of the TRIP project and to the Basque Government Education Department for the financial support to his PhD studies.

References

[1] Chilvers, M., "Fnorb – Version 1.0", Distributed Systems Technology Centre, University of Queensland, Brisbane, Australia, April 1999, http://www.dstc.edu.au/Products/Fnorb/user-guide.html

[2] Dey A.K., Salber D., Futakawa M. and Abowd G. "An architecture to support context-aware applications", UIST '99, 1999.

[3] "GNU Gdbm Database Library", 1999, http://www.polaris.net/docs/gdbm/

[4] Harter A., Hopper A, Steggles P., Ward A. and Webster P. "The Anatomy of a Context-Aware Application", Proceedings of MOBICOM'99, Seattle, August 1999.

[5] Lopez de Ipina D., "TRIP: A Distributed vision-based Sensor System", PhD 1st Year Report, 1999, http://www-lce.eng.cam.ac.uk/~dl231/trip/docs/report.ps.gz

[6] Janssen B., Spreitzer M., Larner D.and Jacobi C. "ILU 2.0alpha14 Reference Manual", Xerox Corporation, 1999, ftp://ftp.parc.xerox.com/pub/ilu/ilu.html#explanation

[7] "Java Naming and Directory Interface (JNDI)" Home Page, 1999, http://java.sun.com/products/jndi/docs.html

[8] Lo. S, Riddoch D, "The omniORB2 version 2.8 User's Guide ", AT&T Labs Cambridge, UK, February 1999, http://www.uk.research.att.com/omniORB/doc/omniORB2/omniORB2.html

[9] OMG, Object Management Group, "CORBA/IIOP 2.2 Specification", February 1998, ftp://ftp.omg.org/pub/docs/formal/98-07-01.pdf

[10] OMG, Object Management Group, "CORBA Services: Common Object Services Specification", September 1998, ftp://ftp.omg.org/pub/docs/formal/98-12-09.pdf

[11] OMG, Object Management Group, "Notification Service – Joint Revised Submission", November 1998, ftp://ftp.omg.org/pub/docs/telecom/98-11-01.pdf

[12] Pascoe J., "The Context Information Service", April 1999, http://www.cs.ukc.ac.uk/people/staff/jp/cis/index.html

[13] Schilit B., Adams N., and Want R. "Context-Aware Computing Applications ", Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, CAIEEE Computer Society, December 1994.

[14] Smith, D., Vinoski, S., "Overcoming Drawbacks in the OMG Event Service", SIGS C++ Report magazine, June 1997

[15] Telstra Corporation Limited, Australia , "Pmw Python megawidgets", June 1999, http://www.dscpl.com.au/pmw/

[16] Want R., Hopper A., Falcão A. and Gibbons J. "The Active Badge Location System", ACM Transactions on Information Systems, Vol. 10, No. 1. 91-102, January 1992

[17] Ward A., Jones A. and Hopper A. "A New Location Technique for the Active Office", IEEE Personal Communications, October 1997, pp. 42-47

[18] Werb J. and Lanzl C. "Designing a positioning system for finding things and people indoors", IEEE Spectrum, September 1998, pp.71-78.