A DISTRIBUTED SOFTWARE ARCHITECTURE FOR GPS-DRIVEN MOBILE APPLICATIONS

Thomas G. Dennehy†
Environmental Research Institute of Michigan
Ann Arbor, MI 48113-4001

ABSTRACT

The unique requirements of voice recognition can shape a software architecture in many ways that have proven effective for mobile and distributed applications. We show in this paper that extending the voice recognition model of translating utterances into sentences to include translating a variety of real-world events into a command protocol can create an architecture whose components operate identically on hand-held devices, man-portable or vehicle-borne units, notebook, or desktop computers. SANSE, a portable navigation and geographic information management system having several redundant user interfaces, is described. In SANSE a collection of distributed *Interactors* translate events—spoken words, input from GPS hardware, timers expiring, input from files or communication links, and direct manipulation actions—into SANSE commands that are sent to one or more *Receivers*, which can execute commands without regard to their source. The complete operation of this system can be captured in vocabulary of less than 70 words, small enough to provide speaker-independent operation yet rich enough to be broadly applicable. The architecture can be extended by adding new Interactor types without affecting the operation of the baseline system.

1. Introduction

SANSE is a software architecture for GPS-driven mobile applications that developed from a simple yet challenging concept: to build a portable navigation and geographic information management system with two completely redundant user interfaces: direct manipulation and voice-activated. The unique requirements of voice recognition shaped the SANSE architecture in a number of ways that proved effective when configuring systems for stand-alone or networked operation. Components of SANSE-based systems can be deployed on hand-held devices, man-portable or vehicle-borne units, notebook, or desktop computers.

For a command language to be effective, it must satisfy a number of criteria [1]:

- *Expressiveness* – The language must provide complete access to the capabilities of the system.
- *Expressiveness of Intent* – The vocabulary must be precise enough, but the user should not be overburdened with expressing his intent. Commands should be short—a few words at most—and to the point.
- *Freedom from Detail* – The vocabulary should be interpreted within a general context in order to cut down the detail that needs to be expressed. This should not be confused with *context-sensitive* grammars, which allow a single word to be interpreted in multiple ways depending on the local sentence context. Such word overloading should be avoided in command languages.
- *Principle of Least Surprise* – The commands and vocabulary should be familiar and natural and behave in expected ways. A Geographic Information System (GIS) may have several related definitions of **North**, for example, and although a particular language may choose to recognize only one of these definitions, the language should not redefine **North** to mean what is generally recognized as **South**.

†The author is currently an independent consultant based in Bloomfield Hills, MI. E-mail to `af510@detroit.freenet.org`.

But an effective command language provides not only a convenient means to *use* a system, but also a natural structure around which to *organize* the system, a particularly effective structure for distributed systems. First, by defining system behavior in terms of a well-understood command set, we can effectively decouple the response to a command from the various and often redundant circumstances that can initiate the command. Second, the command set defines the internal protocol of the system, creating abstract interfaces between architectural components so these components will interact identically whether deployed on a common platform or distributed across a hardware network. Finally, short commands make effective use of inter-process, packet, cellular, and other protocols.

In the next section, we describe the core SANSE architecture for a system with direct manipulation and voice activation, and how this model can be extended for a variety of other input sources. The SANSE protocol is then described, followed by a description of a SANSE-based portable navigation system and discussion of future directions.

2. Core System Architecture

Creating two redundant user interfaces, voice-activated and direct manipulation, is a difficult problem since these two interface styles communicate very differently.

Voice recognition hardware translates utterances into one of several forms, the most common of which are: *isolated words*, where a token representing each individual recognized word is returned to the host; and *connected speech*, where speech energy is interpreted according to a sentence grammar loaded onto the hardware, returning legal sentence structures. To accommodate both styles, SANSE chose sentences as the basis for the protocol between the **Voice Interactor** and the SANSE core. If an isolated word recognizer were chosen, it would be the responsibility of the Voice Interactor to assemble the words into legal sentences.

Visual interface toolkits have various styles of communication. User input can cause events to be posted (X Windows), messages to be sent (Microsoft Windows) or callback function to be invoked (Xt toolkit, Motif widgets). Complete redundancy between the two user interfaces required the **Screen Interactor** to relate to the SANSE core in the same way as the Voice Interactor, making the callback structure not feasible. While events or messages are a valid basis for communication, they didn't match the natural output of the Voice Interactor. The Screen Interactor was therefore designed to translate direct manipulation actions into sentences.

With outside world interfaces translating external events into sentences, the core SANSE component—the **Receiver—**was designed to accept and interpret the SANSE command protocol (Figure 1). From the Receiver's perspective, SANSE operation is a stream of commands that can be executed without regard to the circumstances that created them. From the user's perspective, there is no difference between speaking the words `PAN LEFT` or pressing the corresponding button on screen, allowing voice commands and direct actions to be freely mixed.

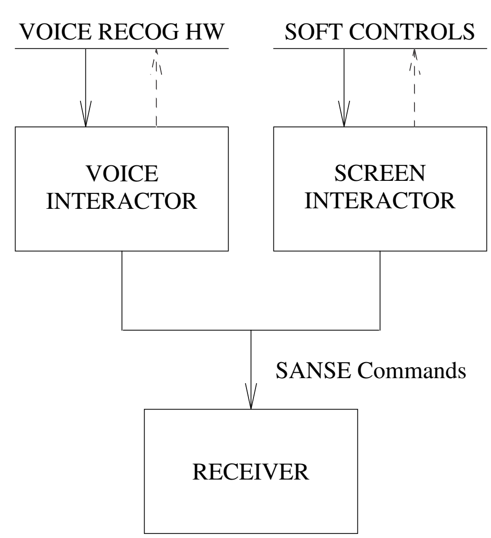

FIGURE 1. INTERACTORS TRANSLATE EXTERNAL EVENTS INTO COMMANDS IN THE SANSE PROTOCOL.

The Receiver executes a command by updating one or more state variables know as *Subjects*. Each Subject has one or more *Views* associated with it that needs to be notified when the Subject is modified. A View is owned by an Interactor, and is simply a representation—visible, audible, or hidden—of one or more Subjects[2]. This Subject/View coupling yields a simple deterministic model for implementing the command set; the complexity of the control structure of the system is independent of the number of commands recognized.

The operation of SANSE can be partitioned into three separable occurrences (Figure 2):.

1) External events are handled by the various Interactors, which translate those events into SANSE commands and send the commands to the Receiver.
2) The Receiver executes commands, modifying one or more Subjects per command executed.
3) When a Subject is modified, it notifies the Views which are currently attached to it, so that the individual Views may update their appearance to reflect the new state of the Subject

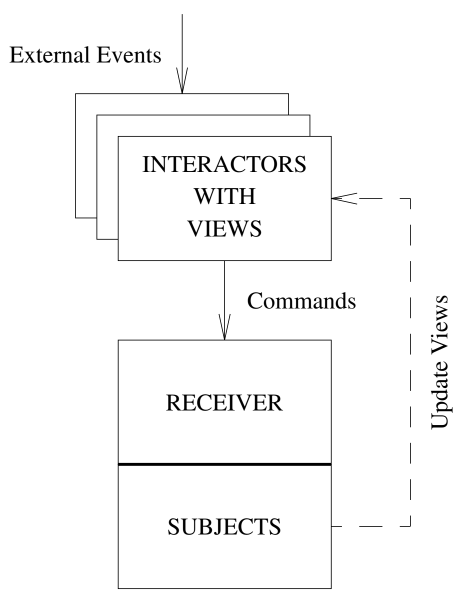

FIGURE 2. COMMAND EXECUTION IS A DETERMINISTIC PROCESS INDEPENDENT OF LANGUAGE SIZE.

View update is a different process from the completely local graphic occurrences intended to provide interactive feedback. For example, when a "soft" button is pressed, its appearance will be altered so as to inform the user that the action has been registered—the shading of its borders may invert, for example—but that action reflects only that a particular button was pressed, not that a particular SANSE command was executed as a result.

To illustrate, an on-screen button may cause a `PAN LEFT` command to be sent when pressed, but saying "Pan Left" should not cause the same visual feedback, as though some invisible hand had pressed the button. However, panning left will update any View that is tied to the Subject representing the Forward direction, and that update will occur independent of whichever Interactor sent the command.

The Interactor model can be extended to any structure that translates physical events into SANSE commands to be interpreted by the Receiver. Three more Interactors have proven immediately useful:

- **GPS Interactor**, which translates real-time global positioning information into SANSE commands.
- **Trap Interactor**, which originates SANSE commands in response to elapsed time or distance traveled;
- **Remote Interactor**, which relays SANSE commands received over remote links or read from files.

The Interactors model the redundancy of the user interface—silent operation or hands-free operation—as well as the independence of the interface components. Given this redundancy and independence, different SANSE systems with various combinations of Interactors can be configured, and the extensibility of the system is well defined. New capabilities, may be added to the system without affecting its present operation by defining a new Interactor to translate new types of events into commands to send to the Receiver, extending the SANSE command vocabulary if necessary.

From the Receiver's perspective any SANSE Interactor is a drop-in replacement for any other Interactor, enabling consistent operation in both stand-alone and distributed configurations. For example, SANSE would operate identically as a self-contained portable navigation system receiving input from an on-board GPS receiver (through the GPS Interactor) or as a desktop tracking system receiving position information from one or more mobile systems (via a Remote Interactor). The Interactors and the Receiver communicate through an abstract interface that can be implemented using a variety of physical channels and protocols[3].

Because Interactors operate independent of one another and independent of the Receiver, SANSE systems can be deployed in stand-alone or networked configurations using a wide variety of hardware components.

- A simple field data collection application can be hosted on a hand-held device using only the GPS and Trap Interactors, operating either in batch mode or in real-time communication with a base station via radio or cellular links. (SANSE's command protocol is well-suited to new packet cellular protocols like CDPD.)
- Portable systems incorporating voice response and GIS displays have been hosted on notebook computers outfitted with single-board peripherals.
- Shadow systems (where mobile system A reports its position to desktop or mobile system B) have been deployed with both systems A & B having full display capabilities.

Advances in CPU power, PCMCIA packaging, and storage capacity will make such self-contained SANSE systems no larger or heavier than the notebook computers hosting the software.

3. The SANSE Protocol

SANSE's command protocol has two representations, an internal packet format, and an ASCII equivalent. The ASCII representation of a command is a sequence of fields separated by semi-colons and terminated by a newline.

```
Org;Mnemonic;C;Keyword;Data;T_Sent;T_Rec
```

The `Org` identifies the command as user-generated (`U`) or system-generated (`S`). Each command `Mnemonic` can have optional `Keyword` and/or `Data`. Data representations for geographic positions, headings, GPS status packets, and numeric choices have been devised; others can be easily added. The `T_Sent` (Time Sent) is supplied by the Interactor originating the command; the `T_Rec` (Time Received) is inserted by the Receiver.

There are two mechanisms for repeating command execution. SANSE will repeat once the last user command executed whenever the Receiver gets the `MORE` command, or will continuously repeat the last user command when the Receiver gets the `CONTINUE` command. The continuation process repeats until the next user command is received. System commands (new GPS location or status information, for example) can be executed without interrupting continuation.

The `C` (Continuation) field of the command protocol has proven valuable for cutting down the communication load between the Receiver and Interactors. Placing the string "ING" in the `C` field is a request for immediate continuation. Thus, if a button owned by the Screen Interactor is intended to provide sustained operation, it can send a command with the Continuation field set when the button is pressed, and send a `STOP` command when the button is released. The communication load is therefore independent of the amount of time the button is depressed, and the controls operate effectively in networked configurations. The Voice Interactor uses the present participle form of certain commands to request continuation: "Panning Left" as opposed to "Pan Left."

Thus, the ASCII representation of the user command `PAN LEFT` would be

```
U;PAN;;LEFT;;;
```

while `PAN LEFT CONTINUE` would appear as

```
U;PAN;;LEFT;;;
U;CONTINUE;;;;
```

but could be appreviated as

```
U;PAN;ING;LEFT;;;
```

Sample content for the data field is illustrated by a choice command like `USE 2`:

```
U;USE;;C 2;;
```

The internal representation of this protocol is fixed-length packets; the packet size is determined by the size of the largest data element it can contain, currently 20 bytes. The unused packet space in commands that contain only keywords or shorter data is more than compensated by avoiding the overhead of sending and receiving data-dependent variable-length packets. SANSE components residing on the same host almost always use the internal representation for routing commands; uncoupled components can choose the ASCII or internal format as required.

The Receiver maintains a history file of all commands executed, with commands stored in their ASCII representation. History files can be replayed through the Remote Interactor. During replay, the Remote Interactor can reproduce or accelerate the relative gaps between commands represented by their individual `T_Rec` stamps.

4. A SANSE Vocabulary for GPS/GIS Applications

Plates 1 and 2 following the text of this paper illustrate a SANSE-based portable system combining navigation and geographic information display with multimedia field data collection and review. This system was implemented with a command set of 33 operations and a total vocabulary of 70 words, a vocabulary small enough to provide speaker-independent voice response. This section describes the operation of that system and its vocabulary.

4.1 Perspective commands. These commands alter the *Field of View*, the region of the earth's surface represented by the GIS display.

Mnemonic	*Argument*
TRACK(ING)	Direction
PLACE	Location or KnownPosition
PAN(ING)	LEFT or RIGHT
LOOK	NumericHeading, Direction, or KnownPosition
TIGHTEN(ING)	
WIDEN(ING)	
ZOOM	IN or OUT
ENLARGE	
REDUCE	
CONVERGE	

The *View Point*—the center of the Field of View—is typically the position reported by the GPS Interactor, but can be established at an absolute location using the `PLACE` command, which takes as its argument a geographic Location or the keywords `HERE`, representing the location currently reported by the GPS Interactor, or `BACK`, representing a previously stored location (see section 4.4). The Field of View can be moved incrementally by `TRACK(ING)` in any of the four compass directions (`NORTH`, `SOUTH`, `EAST`, `WEST`) or `FORWARD`, `BACKWARD`, `LEFT`, or `RIGHT` relative to the *View Heading*. The View Heading is typically the current heading reported by the GPS Interactor, but can be rotated by `PAN(ING) LEFT` or `RIGHT` or positioned at an absolute heading using the `LOOK` command.

The extent of the Field of View (the *scale* of the display) can be changed using the `ENLARGE` or `REDUCE` commands. The degree of enlargement of reduction is controlled by the View Finder, whose size is controlled by the `TIGHTEN` and `WIDEN` commands. `ZOOM IN` makes the View Finder as small as it can be; `ZOOM OUT` removes it from the screen. Finally, the CONVERGE commands restores the display scale to the natural scale of the data being viewed.

4.2 Composition commands. These commands manipulate data sets shown in the Field of View;.

Mnemonic	*Argument*
USE	Choice or NONE
WITH	Choice or NONE
ADD	Choice or ALL
REMOVE	Choice or ALL
HIDE[a]	
SHOW	

a.Equivalent to REMOVE ALL.

The display model combines a raster-based underlay image with vector or symbol-based overlays (annotation). The underlay is a composition of two classes of data: *backgrounds* and *transparencies*. Backgrounds might be scanned maps or satellite photos, while transparencies include land use maps, elevation maps, or related data sets. Although a single background or single transparency could function as the underlay image, there are a number of background/transparency combinations that make tactical sense. The various categories of annotation are rank-ordered by priority, and the enabled overlays shall be drawn in reverse order of priority, lowest to highest.

The `USE` command specifies the background data set to use, or `NONE`. The equivalent command for the transparency is `WITH`. `ADD` and `REMOVE` manipulate layers of the overlay; `HIDE` turns off the current overlays; `SHOW` restores them. Background, transparency, and overlay choices can be always specified by number, and a number of common types of date (`MAP`, `PHOTO`, `TRACE`) have been assigned keywords in the vocabulary. Future versions of the system may support loading customized vocabularies to represent specific data sets.

4.3 Screen Management commands. These commands interact with the window management system through the Screen Interactor;.

Mnemonic	*Argument*
OPEN	Window
WHERE AM I	
CLOSE	Window
RAISE	Window
MOVE(ING)	Direction
BEFORE	
NEXT	

`OPEN` and `CLOSE` can be used to configure the display. The window types recognized are:

- `VIEW` – A window showing the Field of View, along with controls for opening other windows;
- `SCALE` – showing the current map scale, along with controls for changing scale and manipulating the View Finder;
- `COMPASS` – showing position, heading, and status information reported by the GPS Interactor;
- `KEY` – showing the current composition of the Field of View, along with controls for manipulating backgrounds, transparencies, and overlays.
- `POINT` – showing the View Point and View Heading, along with controls to manipulate them.
- `MARKER` – showing information about user-defined markers (see next section).

`WHERE AM I` is a natural equivalent to the command `OPEN COMPASS`.

Most windows are referenced by their keyword alone, but `MARKER` windows are referenced by name and number (“Open Marker 4”) or the most recent Marker if the number is omitted. The user can chain through the entire list of Markers once a Marker window is open using the `BEFORE` and `NEXT` commands. This same feature could be extended to other kinds of windows representing data maintained in lists.

The `RAISE` command brings a particular window to the top and makes it the current window. Although a command is provided for `MOVE(ING)` the current window `UP`, `DOWN`, `LEFT`, or `RIGHT`, using this command is admittedly far less convenient than using a pointing device.

No attempt is made in this command subset to provide access to all the features of a particular window management system or toolkit.

4.4 Action commands. These commands report data, mark locations, and initiate other miscellaneous actions.

Mnemonic	*Argument*	*Qualifier*
MORE		
CONTINUE		
STOP		
REF		
UNREF		
MARK		SYSTEM or USER
GPS_FIX	GPS Fix	SYSTEM
GPS_STATUS	GPS Status	SYSTEM
CHECK		SYSTEM or USER
QUIT		SYSTEM or USER

As previously discussed, `MORE` repeats one the last user command executed. `CONTINUE` repeatedly executes the last user command until the next user command is received. `STOP` interrupts continuation without executing another command. The `REF` command saves the current View Point and View Heading; these values can then be accessed through the keyword `BACK` (as opposed to `HERE`). `UNREF` clears these values.

System-initiated commands are distinct from user commands in that system commands do not interrupt continuation, but instead have their execution interleaved with continuation. `GPS_FIX` and `GPS_STATUS` are System commands that relay position, heading, and status reports from a GPS receiver. The System command `MARK` is initiated by the Trap Interactor periodically to save the current GPS position and heading on a *Trace* of travel. The interval between Trace point can be time-based, distance-based, or a combination. The other System commands are `CHECK`, to run a self-test, or the self-evident `QUIT`; both these commands can also be user-initiated.

One other System command can also be user-initiated. A user can leave a `MARK` at the current View Point, with an option to annotate that mark with data exchanged with other programs. SANSE applications have used Markers containing spreadsheet data, CAD drawings, or audio recordings (an example of an audio Marker is shown in Plate 2). A Marker file with annotation can be preloaded into SANSE, enabling SANSE to be used in the field to update spatial database information in its native format.

5. Discussion

SANSE is written in the C++ language and used primarily on computers running the Microsoft Windows operating system (Release 3.1 and later). SANSE systems can exchange data with other programs through the Microsoft OLE protocol, with SANSE acting as the OLE client. SANSE can be ported to other operating environments, as it incorporates no proprietary non-standard language features and processes only seven Windows messages in the course of its operation.

Although originally written for use in mobile GPS/GIS applications, the SANSE architecture provides a robust general model for mobile and distributed systems by: 1) defining system behavior in terms of a well-understood command set; 2) effectively decoupling the response to a command from the various and often redundant circumstances that can initiate the command; and 3) creating abstract interfaces between architectural components so these components will interact identically whether deployed on a common platform or distributed across a hardware network. This approach pays several benefits:

- Redundancy – In representing the redundancy between elements of the operator interface (silent operation and hands-free operation, for example) the architecture also models the independence of the various elements and how they individually relate to the SANSE core.

- Configurability – Since elements of the SANSE interface are independent, the architecture supports instantiating SANSE with elements selectively enabled/disabled. The command set is easily partitioned; a distributed system can have several Receivers, each recognizing only those commands that can make use of the local platform resources.
- Extensibility – In establishing the allocation of functionality between the SANSE core and various elements of the interface, the architecture supports extending the interface by adding new I/O devices—a video camera, for example—without affecting the operation of the present system.

To write distributed programs, one must be conversant in two distinct vocabularies. First there is the vocabulary of the problem domain, or the *computation model*; software embodying the computation model is called application code. Second, there is the vocabulary of the system domain, or the *coordination model*; software embodying the coordination model is called system code. It has been shown elsewhere that a well-chosen vocabulary for system code can isolate application code from the details of physical process distribution and communication channels, creating distributed programs that may be conveniently ported across different operating environments [3]. Here we have shown that a well-chosen application vocabulary extends this flexibility to the application components, creating systems whose core functionality is isolated from the many redundant sources of its inputs and outputs, and whose diverse components can serve as drop-in replacements for one another to serve a broad range of needs.

6. Acknowledgments and Contact Information

SANSE was designed at the Environmental Research Institute of Michigan (ERIM) in Ann Arbor, MI. The author wishes to acknowledge the many contributors to the project: Orest Mykolenko, Matt Frazer, Lori Sulik, and Linda Spencer for software design; Dave Symanow, Cyrus Wood, and Len Tomko for hardware design and logistics; and especially Ron Swonger for his vision and management support.

Inquiries regarding SANSE may be directed to Jeremy Salinger (`jsalinger@erim.org`) at ERIM, P. O. Box 134001, Ann Arbor, MI 48113–4001.

7. References

[1] Hilfinger, P., *Abstraction Mechanisms and Language Design*, The MIT Press, 1983.

[2] Linton, Mark A., et. al., “InterViews: A C++ Graphical Interface Toolkit,” *Proceedings of the USENIX C++ Workshop*, November, 1987, Santa Fe, NM.

[3] Dennehy, T. G., “Class Libraries as an Alternative to Language Extensions for Distributed Programming,” *USENIX Symposium on Experiences with Distributed and Multiprocessor Systems III (SEDMS III)*, March 26–27, 1992, Newport Beach, CA.