Saphira Robot Control Architecture

Saphira Version 8.1.0

Kurt Konolige
SRI International

April, 2002

1 Saphira and Aria System Overview

Saphira is an architecture for mobile robot control. Originally, it was developed for the research robot Flakey[1] at SRI International, and after being in use for over 10 years has evolved into an architecture that supports a wide variety of research and application programming for mobile robotics. Saphira and Flakey appeared in the October 1994 show *Scientific American Frontiers* with Alan Alda. Saphira and the Pioneer robots placed first in the AAAI robot competition "Call a Meeting" in August 1996, which also appeared in an April 1997 segment of the same program.[2]

With Saphira 8.x, the Saphira system has been split into two parts. Lower-level routines have been re-organized and re-implemented as a separate software system, Aria. Aria is developed and maintained by ActivMedia Robotics. It is a production-level system for robot control, based on an extensive set of C++ classes. The class structure of Aria makes it easy to expand and develop new programs: for example, to add new sensor drivers to the system.

The Saphira/Aria system can be thought of as two architectures, with one built on top of the other. The *system architecture*, implemented entirely in Aria, is an integrated set of routines for communicating with and controlling a robot from a host computer. The system architecture is designed to make it easy to define robot applications by linking in client programs. Because of this, the system architecture is an *open architecture*. Users who wish to write their own robot control systems, but don't want to worry about the intricacies of hardware control and communication, can take advantage of the *micro-tasking* and *state reflection* properties of the system architecture to bootstrap their applications. For example, a user interested in developing a novel neural network control system might work at this level.

On top of the system routines is a *robot control architecture*, that is, a design for controlling mobile robots that addresses many of the problems involved in navigation, from low-level control of motors and sensors to high-level issues such as planning and object recognition. Saphira and Aria share the control architecture duties, with Aria providing the basic elements of action and sensor interpretation. Saphira's contribution to the control architecture contains a rich set of representations and routines for processing sensory input, building world models, and controlling the actions of the robot. As with the system architecture, the routines in the control architecture are tightly integrated to present a coherent framework for robot control. The control architecture is flexible enough that users may pick among various methods for achieving an objective, for example, choosing between a behavioral control regime or a more direct control of the motors. It is also an *open architecture*, as users may substitute their own methods for many of the predefined routines, or add new functions and share their innovations with other research groups.

In this section, we'll give a brief overview of the two architectures and discuss the main concepts of Saphira and Aria. More in-depth information can be found in the documentation at the SRI Saphira web site (**http://www.ai.sri.com/~konolige/saphira**) and ActivMedia Robotics' (http://www.activrobots.com/SOFTWARE) website.

1.1 System Architecture

Think of the system architecture as the basic operating system for robot control. Figure 1-1 shows the structure for a typical robot application. Saphira/Aria routines are in blue, user routines in red Saphira/Aria routines are all micro-tasks that are invoked during every synchronous cycle (100 ms) by Aria's built-in micro-tasking OS. These routines handle packet communication with the robot, build up an internal picture of the robot's state (Aria), and perform more complex tasks, such as navigation and sensor interpretation (Saphira).

---

[1] See **http://www.ai.sri.com/people/flakey** for a description of Flakey and further references.

[2] A write-up of this event is in *AI Magazine*, Spring 1997 (for a summary see **http://www.ai.sri.com/~konolige/saphira/aaai.html**).

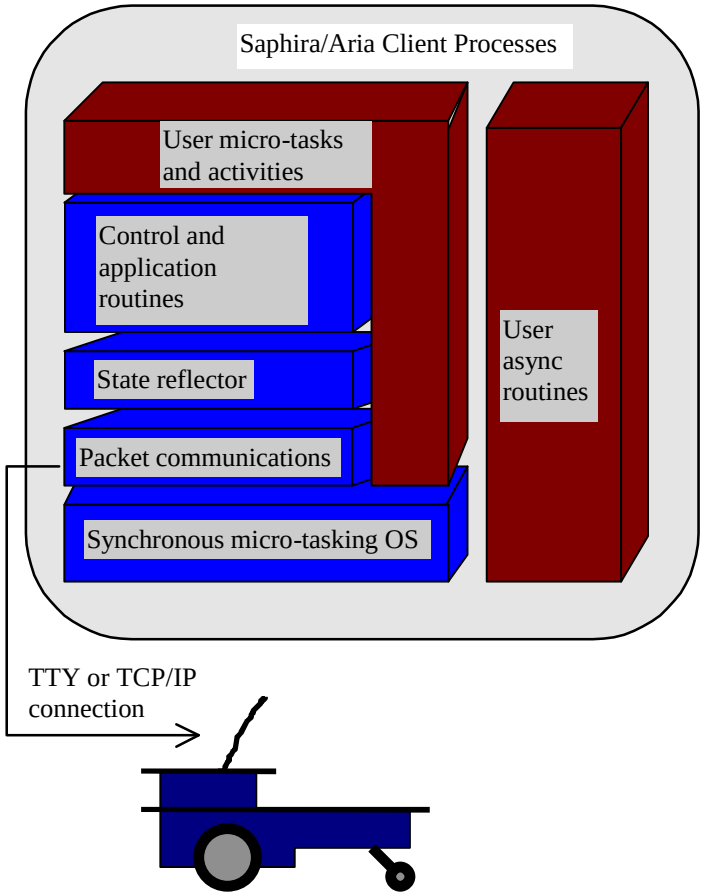

Figure 1-1 Saphira/Aria System Architecture.

Blue areas represent routines in the Saphira/Aria libraries, red routines are from the user. All the routines on the left are executed synchronously every 100 ms. Additional user routines may also execute asynchronously as separate threads and share the same address space.

1.1.1 Micro-Tasking OS

The Saphira/Aria architecture is built on top of a synchronous, interrupt-driven OS. Micro-tasks are finite-state machines (FSMs) that are registered with the OS. Each 100 ms, the OS cycles through all registered FSMs, and performs one step in each of them. Because these steps are performed at fixed time intervals, all the FSMs operate synchronously, that is, they can depend on the state of the whole system being updated and stable before they are called. It's not necessary to worry about state values changing while the FSM is executing. FSMs also can take advantage of the fixed cycle time to provide precise timing delays, which are often useful in robot control. Because of the 100 ms cycle, the architecture supports reactive control of the robot in response to rapidly changing environmental conditions.

The micro-tasking OS involves some limitations: each micro-task must accomplish its job within a small amount of time and relinquish control to the micro-task OS. But with the computational capability of today's computers, where a 500 MHz Pentium processor is an average microprocessor, even complicated processing such as the probability calculations for sonar processing can be done in milliseconds.

The use of a micro-tasking OS also helps to distribute the problem of controlling the robot over many small, incremental routines. It is often easier to design and debug a complex robot control system by implementing small tasks, debugging them, and them combining them to achieve greater competence.

1.1.2 User Routines

User routines are of two kinds. The first kind is a micro-task, like the Saphira/Aria library routines, that runs synchronously every cycle. In effect, the user micro-task is an extension of the library routines and can access the system architecture at any level. Typically the lowest level that user routines will work at is with the *state reflector*, which is an abstract view of the robot's internal state.

Saphira/Aria and user micro-tasks are written in the C++ language, and all operate within the same executing thread, so they share variables and data structures. User micro-tasks have full access to all the information typically used by Saphira/Aria routines.

Although user micro-tasks can be coded directly as FSMs in the C++ language, it's much more convenient to write *activities* in the Colbert language. The activity language has a rich set of control concepts and a user-friendly syntax, both of which make writing control programs much easier. Activities are a special type of micro-task and run in the same 100 ms cycle as other micro-tasks. Activities are *interpreted* by the Colbert executive, so the user can trace them, break into and examine their actions, and rewrite them, without leaving the running application. Developers can concentrate on refining their algorithms, rather than dealing with the limitations of debugging in a compile-reload/re-execute cycle.

Because they are invoked every 100 ms, micro-tasks must partition their work into small segments that can comfortably operate within this limit, e.g., checking some part of the robot state and issuing a motor command. For more complicated tasks, such as planning, more time may be required, and this is where the second kind of user routine is important. *Asynchronous routines* are separate threads of execution that share a common address space with the Saphira library routines, but they are independent of the 100 ms synchronous cycle. The user may start as many of these separate execution threads as desired, subject to limitations of the host operating system. The Saphira system has priority over any user threads; thus, such time-consuming operations as planning can coexist with the Saphira/Aria system architecture, without affecting the real-time nature of robot control.

Finally, because all Saphira/Aria routines are in several libraries, user programs that link to these routines need to include only those routines they will actually use. So, a client executable can be a compact program, even though the Saphira/Aria libraries contain facilities for many different kinds of robot programs.

Packet Communications

Aria supports a packet-based communications protocol for sending commands to the robot server and receiving information back from the robot. Typical clients will send an average of one to four commands a second, although the robot server can handle up to 10 or more per cycle (100+ per second) depending on the serial communication rate and the average command packet size. All clients automatically receive 10 or more server-information packets a second back from the robot. These information packets contain sensor readings and motor movement information, among other details.

Because the data channel may be unreliable (e.g., a radio modem), packets have a checksum to determine if the packet is corrupted. If so, the packet is discarded, which avoids the overhead of sending acknowledgment packets and assures that the system will receive new packets in a timely manner. But the packet communication routines must be sensitive to lost information, and have several methods for assuring that commands and information are eventually received, even in noisy environments. If a significant percentage of packets are lost, then Aria's performance will degrade.

For details about Saphira/Aria client-server packets, study the Aria sources or read about its implementation with ActivMedia robots in the *Pioneer 2/PeopleBot Operations Manual*.

State Reflector

It is tedious for robot control programs to deal with the issues of packet communication. So, Saphira incorporates an internal *state reflector* to mirror the robot's state on the host computer. Essentially, the state reflector is an abstract view of the actual robot's internal state. There is information about the robot's movement and sensors, all conveniently packaged into data structures available to any micro-task or

asynchronous user routine. Similarly, to control the robot, a routine sets the appropriate control variable in the state reflector, and the communication routines will send the appropriate command to the robot.

2.1 Control Architecture

The control architecture is built on top of the state reflector (Figure 1-2). It consists of a set of micro-tasks and asynchronous tasks that implement all of the functions required for mobile robot navigation in an office environment. A typical client will use a subset of this functionality.

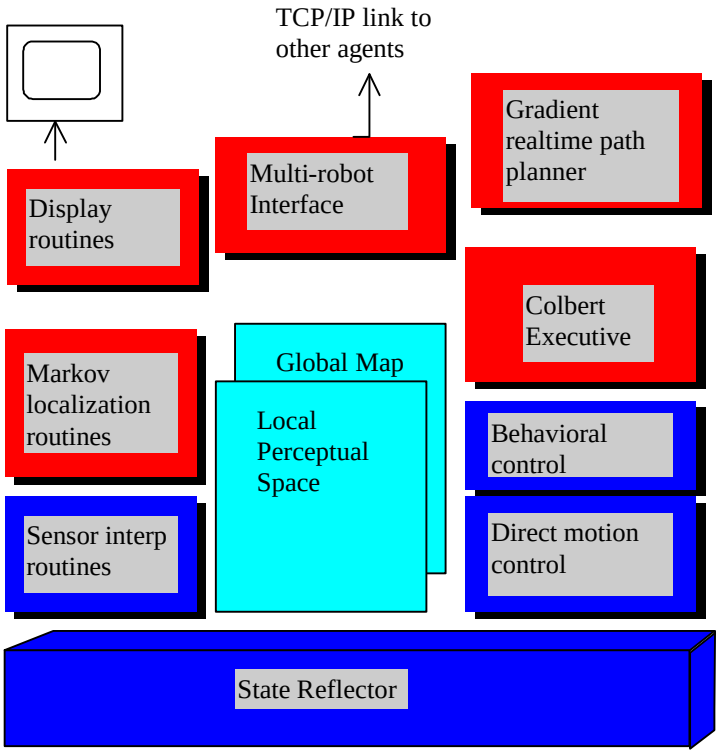

Figure 1-2 Saphira/Aria Control Architecture

The control architecture is a set of routines that interpret sensor readings relative to a geometric world model, and a set of action routines that map robot states to control actions. Markov localization routines link the robot's local sensor readings to its map of the world, and the Colbert Executive sequences actions to achieve specific goals. The multi-robot interface links the robot to other robots using TCP/IP connections. Aria system is in blue, Saphira in red.

Representation of Space

Mobile robots operate in a geometric space, and the representation of that space is critical to their performance. There are two main geometrical representations in Saphira. The Local Perceptual Space (LPS) is an egocentric coordinate system a few meters in radius centered on the robot. For a larger perspective,

Saphira uses a Global Map Space (GMS) to represent objects that are part of the robot's environment, in absolute (global) coordinates.

The LPS is useful for keeping track of the robot's motion over short space-time intervals, fusing sensor readings, and registering obstacles to be avoided. The LPS gives the robot a sense of its local surroundings. The main Saphira interface window displays the robot's LPS (see Figure2-1). In *local* mode (from the Display menu), the robot stays centered in the window, pointing up, and the world revolves around it. Keeping the robot fixed in position makes it easy to describe strategies for avoiding obstacles, going to goal positions, and so on.

Structures in the GMS are called *artifacts*, and represent objects in the environment or internal structures, such as paths. A collection of objects, such as corridors, doors, and rooms, can be grouped together into a *map* and saved for later use. The GMS is not displayed as a separate structure, but its artifacts appear in the LPS display window.

Direct Motion Control

The simplest method of controlling the robot is to modify the robot *motion setpoints* in the state reflector. A motion setpoint is a value for a control variable that the motion controller on the robot will try to achieve. For example, one of the motion setpoints is forward velocity. Setting this in the state reflector will cause the communications routines to reflect its value to the robot, whose onboard controllers will then try to keep the robot going at the required velocity.

Two *direct motion channels* handle rotation and translation of the robot. Any combination of velocity or position setpoints may be used for these channels.

Behavioral Control

For more complicated motion control, Aria provides a facility for implementing *behaviors* as sets of control rules. Behaviors have a priority and activity level, as well as other well-defined state variables that mediate their interaction with other behaviors and with their invoking routines. For example, a routine can check whether a behavior has achieved its goal or not by checking the appropriate behavior-state variable.

Version 8.x includes several major changes in behavior management. Aria implements a general behavior architecture in which behaviors are C++ objects. The interaction among behaviors is implemented by a *resolver* class. Aria provides several types of resolvers, and the user can define his own additional resolvers for particular applications. Behaviors are now integrated with Colbert activities, so that they appear as the leaves of an executing activity tree.

Behaviors can be turned on and off by sending them signals, either from the interaction window, or from the Activities window.

Activities and Colbert

To manage complex goal-seeking activities, Saphira provides a method of scheduling actions of the robot using a new control language, called Colbert. With Colbert, you can build libraries of activities that sequence actions of the robot in response to environmental conditions. For example, a typical activity might move the robot down a corridor while avoiding obstacles and checking for blockages.

Activity schemas are the basic building block of Colbert. When *instantiated*, an activity schema is scheduled by the Colbert executive as another micro-task, with advanced facilities for spawning child activities and behaviors, and coordinating actions among concurrently running activities.

Activity schemas are written using the Colbert Language. The language has a rich set of control concepts, and a user-friendly syntax, similar to C's, that makes writing activities much easier. Because the language is *interpreted* by the executive, it is much easier to develop and debug activities, because errors can be trapped, an activity changed in a text editor, and then reinvoked, without leaving the running application.

Sensor Interpretation Routines

Sensor interpretation routines are processes that extract data from sensors or the LPS, and return information to the LPS. Saphira activates interpretative processes in response to different tasks. Obstacle detection and surface reconstruction are some of the routines that currently exist; all work with data reflected from the sonars, laser range-finders, and motion sensing.

Localization and Maps

In the global map space, Saphira maintains a set of internal data structures (*artifacts*) that represent the office environment. Artifacts include corridors, door, walls, and rooms. These maps can be created either by direct input from a map file, or by running the robot in the environment and letting Saphira extract the relevant information.

Localization is the process of keeping the robot's global location in an internal map consistent with sensor readings from the local environment. Saphira implements an efficient Markov Localization algorithm for taking information from sonars or laser range-finders, matching it to map structures in the GMS, then updating the robot's position.

Realtime, Optimal Path Planning

Saphira 8.x incorporates a new, efficient method for planning optimal paths in real time. The *Gradient Method*, developed at SRI International, operates with both map artifacts and current sensor information to generate optimal paths that move the robot safely through the environment.

Graphics Display

Displaying internal information of the client is essential for debugging robot control programs. Saphira provides a set of graphics routines that can be called by micro-tasks. A set of pre-defined micro-tasks display information about the state reflector and other data structures, such as the artifacts of the GMS. User programs also may invoke the graphics routines directly to display relevant information.

Multi-Robot Interface

Aria is a multi-robot control system, with a class structure set up to handle multiple instances of robot controllers. Currently, Saphira is oriented towards controlling a single robot. In the immediate future, we plan on providing access to Aria's multi-robot facilities through Saphira.

Additionally, we are working on providing a TCP/IP interface between robot controllers running on different physical robots. This interface will tie together Saphira/Aria clients, enabling them to form a distributed robot control system.