A synchronized system concept and a reference implementation for a robot dog

Csaba Kertész
Vincit Oy
Hermiankatu 3 A, 33720 Tampere, Finland
csaba.kertesz@ieee.org

Abstract

Couple of different system concepts have been emerged in the past decades in order to control the robots efficiently. Usually, the robots have resource constraints on processor, memory and power, therefore, the engineers must optimize the algorithms for these conditions. This paper proposes a concept what can be used to design a synchronized system and utilize the increasing computational power of the computers in these days to extend the capabilities of the robots. A reference implementation is presented for the Sony AIBO robot dog, but the concept can be applied to design other kind of systems, which rely on synchronized operations.

1 Introduction

The software of the robots can be run in three basic ways. First, the algorithms can be implemented in hardware components, which offers the fastest execution, though the debugging and bug fixing can be time-consuming. The walking strategy of the hexapod FOBOT (Vámossy et al., 2004) was programmed in PIC microcontrollers to manipulate the servos with low-latency. Building the integrated circuit and the implementation of the strategy was a long process.

An operating system on the robot is more flexible, and after writing the algorithms in programming languages, the sources can be compiled and transferred for on-board execution. The program runs on the robot, the execution is slower than the hardware-accelerated and the development can be a bit complex to debug the algorithms. The Willow Garage's robot platform (PR2) has a high computational potential with two Quad-Core processors in two on-board servers. Maitin-Shepard et al. (2010) concentrated on the towel folding problem with this robot and achieved notable results.

When the computational resources are restricted on the robot, the sensor data (camera image, IR) can be transferred to a remote computer and processed there. Debugging and maintaining the software stack on the computer may be an easier job, but an obvious disadvantage is the high latency of the responses back to the robot. Geusebroek and Seinstra (2005) streamed the camera image of AIBO and used grids of computers around the world for object recognition.

The projects seldom follow these clean principles in the real life, but a mixture is applied for each case. The Universal Robotic Body Interface (URBI) provides both the on-board and the remote execution with the same programming interface (Baillie, 2005). Remote objects can be plugged in on demand for debugging purposes or a distant computer can be used for expensive computation. The URBI runs on a variety of robots like AIBO or Nao, which is used in the Humanoid League of the RoboCup nowadays (Hester et al., 2010).

The advanced robots in the research are expensive, therefore, the development should minimize the run-time of these robots. The simulation environments, like Webots (Hohl et al. 2006), can replace the robots in a certain amount of development time, but the imitations of the real environments are not perfect. The Webots can save the 3D animation of the simulation, however, it can not replay the same situation with the program of the robot if the physics of the simulated environment contains variable conditions; the butterfly effect can make impossible to reproduce these situations. Various challenges can extend the development time with a robot: abnormal termination of the on-board software (e.g. segmentation fault), wrong interpretation of the sensor input (e.g. false object detections) or problems with the locomotion of the robot.

There is no perfect solution, but the new system concept of this paper offers help to overcome some difficulties with robots. For instance, any change in
the source codes can lead to unexpected effects in the behaviour of the robot. The concept can record the incoming data in the system and simulate the situation later. This approach can be used to test the changes in the software and mistakes in the development can be fixed without running the robot. Switching the obsolete test records to new can be done by the simple recording interface API in the concept.

The author's purpose with the proposal fits into the remote execution for robots with resource constraints, but it can also be applied in on-board software stack. The usual frameworks are robot and technology dependent, but the proposal lets the developer to use the framework for any robot or other purposes need synchronous operations. On the other hand, the disadvantage of this approach is the lack of the robot interface, it must be implemented from scratch completely.

The proprietary programming environment of AIBO, the Open-R (Roci et al, 2004), offers platform dependent support for remote execution of the programs for the robot and the on-board run provides only a console via wireless connection for debug messages and a restricted backtrace function in the case of an abnormal termination on the robot. The concept in this paper has a chance to deploy the same software stack to multiple operating systems (Linux and Windows) and let the developer to choice the favourite development/debugging tools in the platform, separately from the robot. Only the C++ language and the MinGW environment under Windows force some restrictions on the free choice.

Similar to SPQR-RDK (Farinelli et al, 2006), the elements can be added and removed from the working system of the concept thanks to the modularity of the elements as well as the data flows between the elements have a standardized form and the computations are synchronized what is an important feature of the neurons' data processing in the human brain (Schoppa, 2006).

The reference implementation, described after the concept, can utilize multiple processor cores or computers to increase the processing power and it uses the Urbiscript to control the body of the robot that helps the re-usability of the developed algorithms, because the URBI is available for multiple robots beyond AIBO. The next section details the concept.

2 Synchronized system concept

2.1 Overview

The system gets input data from outside and processes it. The internal behaviour of the system is (hopefully) deterministic and reproducible later by restoring its initial state and reloading the recorded incoming data. Acting as a scheduler, the synchronicity of the system components is provided by periodical beat signals, using the signal-slot method of the Qt (Molkentin, 2007). The basic definitions:

- **Surroundings**: Everything outside the system.
- **Inflow**: The raw data coming from the surroundings to the system.
- **Context**: It creates, maintains the low-level behaviour of the system and hides from the developers.
- **Normal element**: An entity of the system, which implements data processing functions. It may have input and output.
- **Source element**: The entry point of the inflow from the surroundings to modify the states of the internal components. It receives for example images, translates into a data model of the system and sends to the interested elements.
- **Feeding**: A normal or source element sends output to the interested parties.
- **Sink element**: The sinks receive data from the other elements and represent back to the developer (e.g. playing sounds on speakers). They have important role in the debugging and can have graphical interface to view data.
- **Debug window**: A special sink element with GUI to debug the circulating data in the system. It has a plug-in interface to interpret the future data models.
- **Element data**: The basis of the information exchange between the elements.
- **Data model**: The internal members of the data subclasses. The base data class does not contain variables, but its subclasses declare the new member variables to represent data in the system.
- **Input/output data**: An element data is an output of the sender, however, the same data is an input for the receivers. The sources do not have input, but they can provide any number of output. The sinks get the input data from other elements and they do not output. Any normal element can have any number of inputs and outputs.
- **Session**: One run of the system, interacting with the surroundings.
- **Session storage**: A storage to save the outputs of the source elements and replay a session later based on the records. A plug-in interface interprets the future data models for the storage.
- **Heart**: An entity of the system, which synchronizes the activities of the elements, the data exchange and the session storage with heart beating.
- **Heart beat**: One "tick" of the heart. During a beat, the elements have a certain amount of time for data processing.
- **Activities**: The data processing functions of the elements during a heart beat. The activities mean inflow fetch for the sources, data manipulation for the normal elements and data representation for the sinks.

- **Components**: The common phrase for the internal parts of the system: elements, session storage and heart.

Figure 1 shows the theoretical schema of the concept. The context controls each internal component, the heart synchronizes, the session storage saves or reloads a session and the debug window, as a special sink element, receives the data from the other elements in order to view for the developer. The following subsections detail the context and the components in the system.

![Context Diagram](image)

**Figure 1**: The connections between the components. The debug window and the session storage are grey, because they have plug-in interface. Note that the debug window is shown here separately, hence it is a special element implemented in the core system.

## 2.2 Context

To build a system, an application should create the elements, an instance of the context and start the heart beating. Everything else is automated and hidden from the developer: the context internally creates and manages the heart, the debug window and the session storage. Note that the context does not destroy the elements, it should be done by the application eventually, similar to the creation of the elements.

When the elements are created, usually by declaration, in an application, their constructors register the element instances in the context automatically. Before the heart beating is being started, the elements are also registered by the context in the heart and the session storage is initialized.

To set the properties of the actual session (e.g., saving a session in a specific location on the hard drive), parameters can be passed to the context by the options of the application and there are several setter and getter functions in the public API of the context for the same purpose.

When an application is terminated in an abnormal way (e.g., segmentation fault), it is caught by the context and the session storage is tried to close to avoid data corruption (e.g., invalid video file without index table).

## 2.3 Element data

In the synchronized data flow, the system components need to understand the model of the future data types, therefore, a base class is defined, as the basis of the data exchange. It has some important functions and variables that allow for the components (e.g., elements, session storage) to use data models derived from this class in the higher levels, without prior knowledge.

The solution is a string-to-pointer mapping of the members of the future data models, which must be added by the subclasses in a specific format: name, colon and type. For example, the mapping of a new integer variable called "Power" is "Power:int". The map of the strings and generic pointers to the data members is administrated in the base class.

The definition of the mapping is not vital for every new member variable though the omitted variables will not be visible for the components using the mapping to understand the data models. The assignment of the derived data types work for all, both mapped and non-mapped member variables.

## 2.4 Elements

The duty of the elements in the system is the data processing. The sources fetch the inflow from the surroundings and deliver to the interested parties. The normal elements can receive, process and send data to the others. The sinks have the capability to use a graphical, CLI or other sort of interface to represent some data back to the developer. The type of the element should be declared in its class constructor and kept permanent for the object lifetime.

The multiple instances of the element types have a unique ID (number) what is important to differentiate them. The numbers are set internally when the objects are created and they can be changed, though not encouraged, because the system components rely on (e.g., the session storage stores the output of the sources based on the element type and the ID). The numbering starts from 1 and it is incremented by one for each new instance.

The elements use the subclasses of the base data class for the data exchange. Each data flowing in the system must have a unique name, composed in the
form: name, colon, type. For instance, "Sensors:SensorData" means a data named "Sensors" with the type "SensorData" derived from the base data class. The inputs and the outputs must have corresponding member variables and declared name-to-pointer mappings to the variables in the constructors of the sender and the receiver elements.

By default, the inputs and the outputs of the elements are automatically connected by the context based on their names right before starting the heart beating; the elements are informed about the interested parties regarding a specific output. This step can be done manually, but not recommended. The actual feeding of the receiver elements is done in each beat behind the scenes.

Some inflows from the surroundings can not be available all cases, therefore, the sources have a temporary list of the actual outputs at the end of each heart beat. The sources get the raw data from the surroundings during a beat, turn into the data models of the outputs, fill the actual output list and inform the session storage to save the outputs if the session is being recorded. When the next beat starts, the heart commands the source elements to feed the registered elements with the available data, nevertheless, the implementation of the receivers should take into account that the input data is not always provided. The normal elements act in the similar way, they add a data to their actual output list if it is available.

The activities are not the same for each element type. The sources fetch and transform the inflow, the sinks update a graphical interface with new data or play sound on the speakers, however, the normal elements process their received input and send output to other elements. Let us assuming 10 Hz beating, there are 100 ms to perform actions by the elements and it is not enough for heavy computation. The elements should be designed in a proper way to avoid the blocking of the heart beating, because a beat ends in practice if every element sent a “finished” signal to the heart about the completed activities of the current beat. The heart warns the elements if the expected duration of the actual beat is up, but they are not forced to stop their activities.

At least two design principles can be used to resolve this issue:
- Divide the algorithm into smaller steps and calculate one smaller step during a beat.
- Use a separate thread/processor core for the element, do the calculations in the background in an idle continuously and get new input data/propagate the results through the heart beating.

Normally, the expectations for the source elements are the fast data fetch and providing the latest available inflow, therefore, the design of the sources should consider the fetching at the latest possible moment before the end of a beat, because the data sent to the elements in the begin of the beats is fetched during the previous beat. If the source knows the expected length of the current beat (e.g. 150 ms) and an estimation for the time needed to fetch the inflow (e.g. 40 ms), the activities of the source element can be scheduled to the latest possible time (150 ms-40 ms = 110 ms). This delay can be refined for every beat.

2.5 Heart

A component, which synchronizes the activities of the elements and acts as a scheduler in the system. One smaller computational period corresponds to a beat in the terminology. A new beat is started if the previous is finished by all elements, however, the limit is not hard-coded for the duration. Any element can request a limit for the next beat, the heart receives that and sets a duration limit for the next beat. Using a beat frequency between 1-15 Hz is a good intention for a robot to get sensor data often and leave enough time for the data processing. If the beat duration time is up then the heart warns the elements to finish their activities, but the elements are not forced directly to stop their activities. The elements can also finish their activities earlier than the limit; a new beat begins immediately. The context can send an emergency signal to exit, but it is reserved for situations when the session dies with a segmentation fault or a similar signal.

The context registers the elements in the heart and the beating is started/stopped by the context as well. The activities of the sources are started in the first beat and all elements are active in the following beats. The reason is to let the sources to fetch the first portion of the inflow to feed the interested elements from the second beat. The beats inside a session are identified by a unique ID started from zero and it is always increased by 1 for the next tick. This ID has a vital role to record and play back the sessions.

On the Figure 2, the normal arrows follow the algorithmic steps during a beat. The heart sends a start signal for the elements and they progress with their tasks. When the data processing is done, the elements send “finished” signals to the heart and the iteration starts again. The dashed arrows indicate the data flow on the diagram, the session storage can load/save the output of the sources or for example a robot interface provides inflow (raw sensor data) for the sources. Eventually, the sources send the data to the receivers and normal elements can send output of their data processing from the previous beat. Note that the robot interface does not part of the core.
system, it is not implemented here; showed solely for demonstration purposes.

The threads are used to parallelize functions to improve the performance (e.g. running independent calculations in separate threads/processor cores decreases the run-time), but the number of the processor cores determines the optimal performance. When a heavy calculation uses a core on maximum level, new computational threads can be run on other available cores, however, if all cores have full load, running a new thread does not boost the performance, because there is no free processor resource any more.

The Qt has a design restriction for the graphical widgets; they can be run only in the main thread of an application. For these two reasons above, the heart applies the following rules for the elements:
- The sink elements with GUI remain in the main thread.
- New, dedicated thread is created for each element with thread option enabled.
- A computational thread is created for all the remaining elements and they are moved there to avoid blocking the user interface.

2.6 Session storage

As mentioned in the previous subsections, the sessions can be saved by storing the output of the source elements in each beat and they can be reloaded later. This is a beneficial approach to simulate the system behaviour without active source elements, whose activities are skipped in that case.

The outputs are stored in a database or on the hard disk and they are identified by the element type, the element ID and the beat number. The element type determines a certain output list, but several element instances can exist in a working system, therefore, it is needed to know, which instance is in question. Additionally, the beat number specifies the output data for a specific beat.

There are two ways to store the outputs of the sources:
- Sql database: It comes into the picture when the data is relative lightweight and contains only basic primitives (integers, floats or short strings). The Sqlite backend of the QtSql is used, which is a fast and efficient choice for a small database in a standalone file.

- Media files: The image and the sound are frequently used in various systems and they are considered here as heavier binary data. The temporal appearance of the image and sound are video/audio streams and it is straightforward to save them into media files.

The session storage must understand the new data types defined in higher levels, therefore, there is a plug-in interface where the handling of basic types for the Sql database can be written and coder/decoder pairs can be added for the binary types.

It is not granted that outputs are available every beat, hence the relevant records are marked as empty in the Sql database. The coders/decoders for the binary types are implemented fully through the plug-ins and thus the plug-in implementations must pay attention for this issue.

The playback of a session is sensitive for the existence of the sources. These elements must be created with the same type and in the same order to identify the exact matches between the living instances and the saved records. A mismatch between the IDs of the present instances and the IDs in the database causes the termination of the session, because heuristics are not implemented in the session storage to resolve this situation. On the other hand, if some sources do not live, but they have records in the database, those records are ignored and it becomes a good option to suppress particular sources from a playback.

2.7 Debug window

The debug window is a special sink element with GUI to debug the data in the system during run-time, which has been developed after reaching the limits of command line debugging. The GUI is a window and the views of the data models are embedded dialogs showed on the Figure 3. They contain textual or graphical representation of the outputs of the elements and each dialog belongs to one data provided by one element. Dragging and zooming the dialogs make easy to change the subject. On the status bar, the current beat ID and the duration of the previous beat are shown.

Similar to the session storage, the debug window needs to understand the data subclasses coming from higher layers. There is a plug-in interface to register the data subclasses and their views.

In fact, the debugging should have low resource consumption to avoid the influence on the observed system. High CPU load of the frequent dialog
updates has been discovered and led to different strategies to minimize this effect. The dialogs can be zoomed in to inspect their content or zoomed out to the background. In the latter state, the dialog is not updated because the observer does not pay attention to it. The remaining active dialogs are updated maximum in 5 Hz frequency that is suitable for a human observer.

Figure 3: An example for the debug window from the reference implementation. Two dialogs are zoomed in, which belong to two data types named Sensors and RobotState.

3 Reference implementation

3.1 Overview

The control of a robot dog, proposed in this section, is divided into two places. The on-board (URBI) control is responsible for the immediate responses and the reference implementation of the system concept, the AiBO+ (http://aiboplus.sf.net), on the computer takes care of the learning, planning and thinking of the robot. It is a good division between the particular tasks, because the computational and power resources are limited on the robot and constant, however, the computers are evolved year by year and it is easy to switch the underlying technology.

The robot dog runs an URBI server and the computer can connect there via a local wireless network. The sensor data is transferred to the computer and commands are sent back to AIBO. In theory, it would be possible to control the robot over the internet, but the high latencies do not make this solution feasible, because the image, the sound and other sensor data must be transferred to the computer in a reasonable amount of time.

The concept uses a specific term, surroundings, for the external world in the previous sections. From the point of view of the robot, the external environment is sensed with its sensors (camera, infrared, temperature), therefore, the meaning of the “environment” is different for the system concept and the robot, because the concept is one part of the robot brain in the AiBO+.

3.2 State machine

Currently, the AiBO+ implements a basic state machine (Figure 4), which takes into account the daylight/lightning changes. The darkness indicates the night for the robot, it lies down to the ground, the heart beating is slowed down and the motors are switched off. If the sunshine lights the environment of the robot, the beating is accelerated back to the normal speed, the motors of the joints are switched on and the robot stands up.

When the robot stands and the motors are not being moved, a static balance strategy is applied. AIBO can keep the posture against hustle, but its motors are not so fast and strong, therefore, this ability is limited.

Basically, the mood of the robot can be happy or angry. Moving the non-functional joints during sleep or picking up from the ground and carrying frustrates AIBO, the pulse number raises and starts to bark furiously until the disturbance is stopped. The temporal transitions between the states are continuous in place of discrete, since some e-hormones are used in the system, similar to their biological counterparts.

The pineal gland maintains the melatonin level in both the human and the animal brain. When the eyes see brightness, the production is reduced; in the case of the darkness, the production is increased. For the robot dog, the high e-melatonin level corresponds to the sleep mode and the low level to the awaken state. The brightness of AIBO's camera image decreases, the darkness increases the e-melatonin level in every beat and the state of the robot makes transitions between sleeping and awaken states.

The happy-angry states simulate simple artificial emotions. In normal conditions, the robot dog is happy and the relevant e-adrenaline level is low.
However, the disturbance of the robot raises the e-adrenaline level, the heart beating is accelerated. At high level, AIBO barks and shows unhappy face expression with the LEDs on its head. Stopping the disturbance gets back to the normal mood.

3.3 Derived element and data types

The AiBO+ uses the following main data subclasses to represent the sensor data and the states of the robot body:
- MImage: Encapsulates one camera image of the robot.
- MSound: Encapsulates the microphone records of the robot.
- MSensors: The subclass can contain the raw sensor data (leg joints, accelerometer, buttons etc.).
- MHormones: The data type of the e-melatonin and the e-adrenaline.
- MRobotState: It holds the robot state extracted out of the temporal sensor data series (legs state, posture, mental state etc.).

Several element types are defined with specific purposes:
- MImageSource: A source, which receives the camera image of the robot and distributes to the other elements.
- MSensorsSource: This source element fetches the raw sensor data of the robot, converts to an internal model and sends to the interested elements.
- MSoundSource: Fetches the recorded audio from the robot.
- MInnerWorld: A normal element, which interprets the sensor data and builds a model of the robot and its environment.
- MRobotMind: A normal element to make decisions upon the internal understanding of the world.
- MBodyControl: Controls the body of the robot. Only this subclass can send URBI commands to AIBO.
- MSoundSink: This sink element plays sound on the speakers attached to the computer.

The AiBO+ also implements the plug-in interfaces for the debug window/session storage and a permanent data storage to save instances of the data subclasses.

4 Conclusion

A new system concept and its reference implementation have been proposed in this paper. The heart beating provides a design principle for the data processing in the systems where the data flow can be standardized and system input can be saved to simulate of the system behaviour again.

The AiBO+ builds a simple system upon the concept and implements a state machine as well as the open-source initiative of the project can encourage other people to develop synchronized systems based on the proposed concept.

Future work can include the extension of the concept to support complicated designs and the reference implementation can also evolve to more complex state machines.

Acknowledgements

Thanks to my family to tolerate my hobby research at home and my company to support some parts of my efforts.

References


