SkRobot: a pseudo-realtime multiplatform framework for robotics agents development*

Giovanni De Gasperis[1,*,†], Daniele Di Ottavio[†], Patrizio Migliarini[1] and Stefania Costantini[1]

[1]Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Universitá degli Studi dell'Aquila, Italy

Abstract

SkRobot is a software platform that simplifies robot development, especially for those with cognitive capabilities. It uses the C++ SpecialK framework as its foundation. It provides active data brokering, distributed storage and processing, and pseudo-realtime synchronisation, enabling efficient communication between system entities. The platform relies on FlowProtocol, a custom protocol that ensures robust binary communication over network channels. SkRobot's architecture is designed for simplicity and efficiency, allowing developers to focus on functions while reducing the influence of system artefacts. This framework enables developers to quickly understand robotic paradigms, including cognitive robotics, and meet various implementation needs efficiently.

Keywords

agents, cognitive robotics, distributed systems, robot development, framework

1. Introduction

The book *"Artificial Intelligence: Foundations of Computational Agents"* by David L. Poole and Alan K. Mackworth [1] presents a model of computational agents relevant to cognitive robotics. An agent is defined as an entity that pursues goals by interacting with and modifying its environment based on sensory inputs and feedback. Agents are classified by their environmental impacts and goal attainment methods. Artificial agents are either reactive or intelligent. Reactive agents respond to stimuli with predetermined actions and are prone to errors in complex scenarios. Intelligent agents autonomously analyse information to make decisions based on context and learned experiences [2]. Agents are structured hierarchically with three main layers: the decision-making level evaluates perceptions; the central level manages control, perception, proprioception, and feedback; and the peripheral level interfaces with hardware, including sensors and actuators.

WOA 2024: 25th Workshop "From Objects to Agents", July 8-10, 2024, Forte di Bard (AO), Italy
https://www.univda.it/woa2024/

*Corresponding author.
†These authors contributed equally.
giovanni.degasperis@univaq.it (G. De Gasperis); daniele.diottavio@gmail.com (D. Di Ottavio); patrizio.migliarini@univaq.it (P. Migliarini); stefania.costantini@univaq.it (S. Costantini)
https://www.disim.univaq.it/GiovanniDeGasperis (G. De Gasperis); https://gitlab.com/Tetsuo-tek (D. Di Ottavio); https://exosoul.disim.univaq.it (P. Migliarini); https://www.disim.univaq.it/StefaniaCostantini (S. Costantini)
0000-0001-9521-4711 (G. De Gasperis); 0000-0002-7824-529X (P. Migliarini); 0000-0002-0877-7063 (S. Costantini)

The physical body of an agent, comprising sensors and actuators, is crucial for interacting with and modifying the environment to achieve goals, positioning it within embodied artificial intelligence [1, 3]. Sensors detect environmental stimuli processed internally to form perceptions. Actuators execute actions based on decisions, with some incorporating feedback devices to monitor and adjust actions [1, 4, 5]. Proprioceptors, or internal sensors, monitor action progress and provide feedback that can modulate actuator activities, essential for rapid response adjustments. Perceptions are multi-stimulus responses requiring efficient data management for real-time responsiveness [1, 6]. Attention mechanisms filter out irrelevant stimuli, enhancing computational efficiency and decision precision. The agent's state is time-dependent, defined by discrete operational phases, with transitions triggered by perceptions, affecting both external actions and internal state. This dynamic interplay of state, attention, and action underlines the adaptive nature of cognitive agents in embodied AI systems. Integrating real-time, parallel, and asynchronous communication is essential in contemporary distributed programming, posing significant challenges and offering innovative solutions through advanced middleware design [7, 8]. In this scenario, we have identified several design solutions and described them below, each with its own strengths and weaknesses; to address some of the weaknesses, we have decided to develop a new solution: **SkRobot** [9].

2. Literature Review and Related Works

Designing a robot requires a multidisciplinary approach, combining systems and network engineering, physical-environmental sciences, and electronics. A robot's operability depends on interacting with an operating system that manages resources like memory, storage, network capabilities, energy, and environmental data, often through peripheral devices. Software control is crucial, with algorithms related to robot's perception and actions. While the high-level programming language is less critical, efficient low-level component design is essential. Developers must integrate multiple complex programs that interface with users or process environmental inputs such as audio or video, recognising elements like objects or sounds. Using pre-built middleware and specialised applications simplifies development. For example, Redis is commonly used for real-time data brokering in distributed systems. Frameworks and SDKs like ROS (Robot Operating System) offer essential tools and libraries, facilitating robotic functionalities without starting from scratch [10]. Though developers often implement many features, ROS acts as middleware, abstracting hardware to manage processes and communication. Its modular architecture allows focused development on navigation, perception, or control. The active ROS community contributes to a repository of software packages, solving common robotics challenges and promoting innovation and efficiency in robotic design.

The evolution of SpecialK [1] involved extensive experimentation with various frameworks, notably the Qt development framework [11]. SpecialK adapted Qt's Signal/Slot paradigm through reverse engineering, surpassing the traditional callback mechanisms used in ROS and creating complex yet manageable connection graphs among class functionalities. However, Qt was eventually deemed unsuitable due to its high commercial costs and event management system, which does not prioritise time — a critical factor for distributed systems with pseudo-

[1] https://gitlab.com/Tetsuo-tek/SpecialK last accessed June 2024

realtime synchronisation. These limitations helped refine SpecialK's programming concepts and paradigms. Additionally, SpecialK integrated programming modalities from other platforms, such as the C++ sketches from the Arduino platform for firmware-oriented microcontroller programming [12]. These sketches, with a setup configuration section and a cyclically called loop section, inspired the design of SpecialK's event manager and the Sk/PySketch engine [2] [3] [4], a Python binding of the flow protocol compatible with Python versions 2 and 3. This integration underscores SpecialK's commitment to developing robust and efficient programming structures for complex robotic and embedded system applications.

3. Methodology

Delving into the design and subsequent implementation, the first significant challenge encountered consists of constructing a nervous system that enables the entity to receive stimuli from its surroundings and perform valuable and logical actions by modifying the state of the environment itself without allowing processes of acquisition and activity to interfere with each other. The main features of the communication layer that connects all the entities making the nervous system are (1) parallelism, (2) functional asynchrony, (3) reactivity efficiency, (4) real-time (as much as possible), and (5) data and events distribution.

Effective robotic design focuses on low-level communication systems for asynchronous input/output distribution, akin to efficient structures in vertebrates. These systems are crucial for both autonomous and reactive high-level decision-making. The design approach should be straightforward, minimising the impact on the host system and maintaining transparency to avoid unnecessary complications. Establishing fundamental principles that clarify related concepts is essential for intuitive understanding, often requiring in-depth theoretical research. Creating new tools from scratch is sometimes necessary to enhance knowledge and simplify the design process. The SkRobot environment, based on the SpecialK framework, exemplifies this approach. It offers a streamlined method for developing robotic systems, addressing various implementation needs in robotics and related fields. SkRobot enables developers to quickly grasp and apply different study cases and solutions, simplifying the study and design process. The concepts learned through development under SkRobot and SpecialK are few and well-defined. Still, they are the cornerstones of the discussion, whatever the implementation you are using: (a) active data-brokering, (b) distributed storage, (c) distributed processing, and (d) pseudo-real-time synchronisation. SpecialK is a C++ framework perfectly compatible with the STL (Standard Template Libraries). Sk essentially enforces the following paradigms: asynchronous and recursive destruction of objects, Signal/Slot interactions between potentially unknown objects, events and pulsing management (ticks, which can be regulated in terms of type and frequency).

The design of robust and lightweight time management applications avoids mutexes or semaphores using asynchronous collaboration, following Sk's paradigms for efficient parallel programming. Similar to the Qt framework, specific scenarios allow single process flow con-

[2]https://gitlab.com/Tetsuo-tek/PySketch last accessed June 2024
[3]https://gitlab.com/Tetsuo-tek/SkRobot/-/blob/main/examples/publisher.py last accessed June 2024
[4]https://gitlab.com/Tetsuo-tek/SkRobot/-/blob/main/examples/subscriber.py last accessed June 2024

currency through the Signal/Slot paradigm. Sk minimises dependencies on external libraries, typically requiring only open-source components like OpenCV-4, PortAudio, FFTW3, Ogg Vorbis, and FLTK for GUI support. Features and dependencies can be toggled via compilation macros, allowing direct inclusion of framework artefacts in the application code and enhancing control over framework changes. The foundational class in SpecialK's hierarchy is *SkFlatObject* [5], providing minimal functionality beyond naming instances. All Sk data structures [6] derive from *SkFlatObject*, designed for simple instantiation and automatic memory management when stack-allocated. Derived classes not using the Signal/Slot mechanism or asynchronous destruction also originate from *SkFlatObject*. The *SkObject* class [7], a derivative, introduces enhancements for efficient programming flows, utilising macros and limited C++ templates[8]. Instances of *SkObject* derivatives should be created with the *new* operator and destroyed asynchronously through *destroyLater()* to prevent runtime errors. This ensures smooth integration with ongoing system interactions and decommissioning objects after careful termination of relationships and activities. Objects are destroyed automatically only in two known cases:

1. the object is set as a child of a parent object, and the last one is destroyed; this occurs for all direct children, as well as recursively for all children of children, following what can be defined as a destruction tree;
2. the event manager terminates its activity and some instances of *SkObject*-derivatives have been instantiated under its control; this occurs when a thread is closed or, more simply, when the application is closed (reverting to the situation at point 1 if it is the case).

For objects within Sk, other than two specific exceptions, it is necessary to use the *destroyLater()* method to ensure the proper release of resources. Without invoking this method, resources remain allocated until the owning *SkEventLoop* [9] manager terminates. This system operates on a pulse or tick basis, a concept that mirrors the operational flow of an Arduino sketch, where activities are driven by the regular execution of the *loop()* function. This approach underpins the design of the SpecialK Python sketches (Sk/PySketch) and the implementation of the flow protocol. In the Sk framework, each thread, including the main application thread, operates under an instance of *SkEventLoop*, which generates ticks at configurable intervals and modes. For optional threads, the tick interval and mode can either be customised or inherit the default settings from the main application thread, allowing for synchronised or individualised thread operations. Each event loop manager emits cyclically three types of ticks with different speeds; the fastest, non-divisible, therefore atomic, also describes the temporal resolution of response (lag) to external and internal solicitations:

- **FastTick** - is the fastest tick provided by the manager, active or passive, depending on the mode set; it represents the maximum processing speed in the thread where the manager resides;
- **SlowTick** - is a passive tick with an interval >= the FastTick interval;

[5]https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkFlat/Core/Object/skflatobject.h last accessed June 2024
[6]https://gitlab.com/Tetsuo-tek/SpecialK/-/tree/master/LibSkFlat/Core/Containers last accessed June 2024
[7]https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/Object/skobject.h last accessed June 2024
[8]https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkFlat/skdefines.h last accessed June 2024
[9]https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/App/skeventloop.h last accessed June 2024

- **OneSecTick** - is a passive tick interval always equal to 1 second.

In the Sk framework, passive waiting for SlowTick and OneSecTick is managed using the *SkElapsedTime*, a nanosecond-resolution timer that counts intervals. Active waiting, in contrast, involves suspending the thread for a less or more precise time using *usleep(...)* or *nanosec(...)*, applicable only in the FastTick mode, which sets the computational cadence for the thread. SlowTick and OneSecTick are used for less frequent operations like monitoring, visualisation, and control. They should not be mixed with FastTick operations that handle more immediate external events, such as network communication, to prevent processing delays. SlowTick and OneSecTick operate as passive clocks relative to the FastTick, with their accuracy dependent on the fast interval's size and regularity. More precise and smaller fast intervals result in better timing for SlowTick and OneSecTick. Conversely, more oversized or irregular fast intervals may lead to variable and non-deterministic timings for these slower ticks. For applications involving multiple threads where some function as producers and others as consumers, it's crucial that consumer threads pulse at a frequency greater than or equal to their associated producer to prevent issues like queue overflows or deadlocks in concurrent environments. If threads share data through non-blocking structures like the *SkRingBuffer*[10], less frequent data acquisition than production may occur, which can be an intentional choice by developers for sampling purposes. It's also vital to monitor the job-time amplitude during tick processing to ensure it doesn't approach the upper limit of the FastTick interval. Exceeding this limit can slow the tick rate, leading to a longer average pulse interval and potential application performance degradation. The cadence typology for FastTick, as a consequence also for SlowTick and OneSecTick, can follow various modes:

- **regular coarse timing** - never equal to or below the required interval (resulting in time loss), used as default and very light as it calls *usleep(...)*, which puts the process in a passively timed pause evaluated by the kernel in this case;
- **pseudo-real-time regular timing** - more than regular average time, slightly more CPU-intensive as it calls *nanosec(...)*, which counts (within the process) nanoseconds based on elapsed machine cycles, thus keeping the process always active within the time window assigned by the Kernel;
- **irregular timing dictated by socket I/O activity** - CPU workload can be very intensive if sockets traffic is significant; in this case, the value set as the FastTick interval corresponds to the maximum wait time on sockets to detect data presence (select); it is an upper limit to the tick interval since it waits at most for the proposed time interval for each existing active socket in the owner thread;
- **irregular timing dictated by GUI activity** - based on the FLTK library event handler, always very light on the CPU (the GUI portion of Sk enabling graphical application development is not described in this document);
- **no timing** - requires a blocking call of any type within the FastTick pulse scope, aimed at slowing it down as it would do when reading from a blocking socket connected that has no data available yet; if not slowed down, this pulsing mode is comparable to a while(1)...

[10]https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkFlat/Core/Containers/skringbuffer.h last accessed June 2024

The Signal/Slot paradigm (Fig. 2) [11] [12], pivotal in programming workflows and evolution from callback functions, dictates a specific interaction flow in applications. Historically, callback functions — originating from C and prevalent in various languages, including ROS — serve as parameters set during initialisation to respond to specific events. A typical example is a graphical interface where a button triggers a predefined function, linking an action (button click) directly to a response through event setup. In this paradigm, a Signal is a method declared in the header file without any associated scope. At the same time, a Slot functions similarly to a standard method but always returns void. Signals can be connected to one or multiple Slots, which could belong to the same or different objects, through the *Attach(...)* functional-macro [13]. The following is an example of *Attach* functional macro related to establishing some Signal/Slot meta-connection:

```
Attach(svr, addedChannel, this, onChannelAdded, SkQueued);
Attach(svr, removedChannel, this, onChannelRemoved, SkQueued);
Attach(eventLoop()->fastZone_SIG, pulse, this, onFastTick, SkDirect);
Attach(eventLoop()->oneSecZone_SIG, pulse, this, onOneSecTick, SkQueued);
```

Listing 1: Attach syntax examples to establish Signal/Slot meta-connections

This connection can be dissolved using the *Detach(...)* method, ceasing the Signal's ability to invoke the Slot after a tick. Below is an example of *Detach* functional macro related to breaking some Signal/Slot meta-connection:

```
Detach(svr, addedChannel, this, onChannelAdded);
Detach(svr, removedChannel, this, onChannelRemoved);
Detach(eventLoop()->fastZone_SIG, pulse, this, onFastTick);
Detach(eventLoop()->oneSecZone_SIG, pulse, this, onOneSecTick);
```

Listing 2: Detach syntax examples to break Signal/Slot meta-connections

Some in-depth information and code snippets are available at SpecialK repository [14].

Signals and Slots must be declared publicly within the class to ensure they are observable, accessible and manageable by the event manager. This is crucial for handling inheritance and interaction across derived types. A protected or private declaration of Signals/Slots will lead to runtime errors during attachment attempts due to visibility restrictions to the manager. To avoid issues with access levels when deriving types that include Signals and Slots, the 'extends' macro is used to ensure public inheritance. This architecture facilitates the synchronisation of activities across different object types and threads without needing mutual exclusion mechanisms like mutexes and wait conditions, streamlining Inter-Object Communication (IOC) and enhancing program responsiveness. The *Attach* and *Detach* operations in the Signal/Slot paradigm are asynchronous, not executing immediately but scheduled for the next pulse by the event manager. Consequently, if a Signal is triggered right after an *Attach*, the connected Slot won't respond until the following tick. *Attach* usually occurs in the object's *Constructor*, and *Detach* automatically at the object's destruction, ensuring stable connections throughout the object's

[11] https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/Object/sksignal.h last accessed June 2024
[12] https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/Object/skslot.h last accessed June 2024
[13] https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/Object/skattach.h last accessed June 2024
[14] https://gitlab.com/Tetsuo-tek/SpecialK/README.md last accessed June 2024

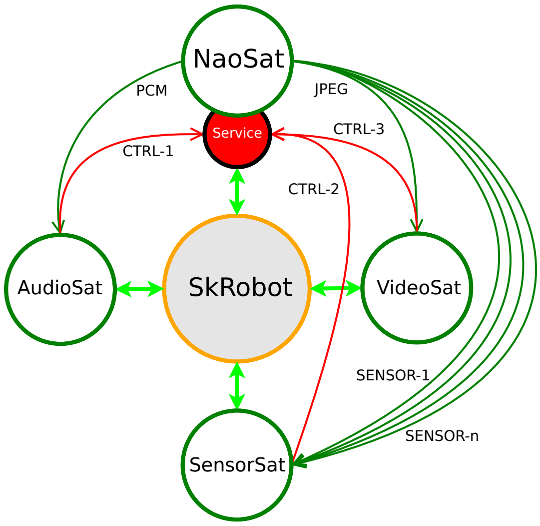

Figure 1: FlowNetwork example graph for Nao base implementation

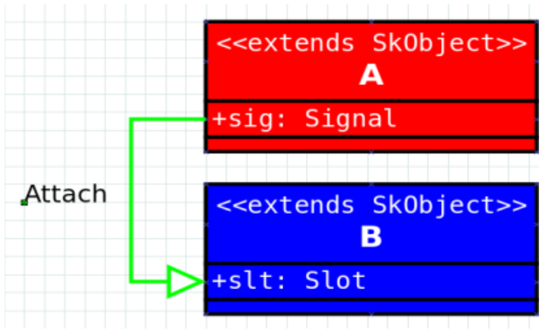

Figure 2: Signal/slot paradigm

lifecycle. Triggering a signal acts like a usual method of calling. If linked to one or more Slots, these Slots are invoked after the Signal is activated, executing their scoped code directly or, when connected in asynchronous mode, in their respective thread as soon as possible. This setup ensures the Signal's immediate trigger with subsequent flexible Slot execution.

The connection mode between Signal and Slot can be of different types:

- **Direct** Slots are invoked directly when the Signal is triggered. This is similar to the Slot method being called to execute the code in the triggering thread where the code that requested the Signal triggering is live. It's important to note that manually invoking a Slot through the method provided by the manager (invokeMethod()) is not direct and will always occur at the next pulse asynchronously.
- **Queued** - Slots are queued for future invocation by the event manager at the next round and in the flow of the owning thread, even when the Signal call comes from another manager, hence a different thread. Triggering a signal connected with queuing mode never blocks the triggering call, even when the signal and slot reside in the same thread; this still implies the asynchronous invocation of the slot connected to the next pulse when the triggering purpose is already closed.
- **OneShot** (direct or queued) - Slots are invoked as illustrated in the previous two points but only once; immediately after the invocation, the disconnection occurs automatically.

A signal can simultaneously connect to multiple slots but never connect to the same slot more than once. Upon triggering, a signal can pass a list of *SkVariant* [15] type arguments used by the invoked Slots. If the call is direct, Slots access pointers to the original values; if queued, they receive argument copies, thus avoiding critical sections and mutual exclusion issues.

[15] https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkFlat/Core/Containers/skvariant.h last accessed June 2024

The *SkVariant* class efficiently encapsulates diverse data types, including primitives, complex structures, and pointers. For threading synchronisation without mutexes or wait conditions, queue-type connections between Signals and Slots from different threads are recommended to prevent deadlocks and efficiency losses due to micro-waits. This paradigm also allows various processing tasks to be isolated across different classes without requiring direct knowledge of each other, maintaining interaction through functional and dynamic runtime meta-links. The SkRobot application, developed in C++, offers a robust platform for managing data flows in autonomous devices, robotic systems and industrial production lines. It supports Input/Output management and custom internal network services, requiring a Unix-like environment (e.g., GNU/Linux or *BSD) and minimal dependencies, adhering to the POSIX standard. This setup ensures SkRobot is a flexible, easy-to-implement solution that can be adapted and modified even during production. Sk defines a communication protocol named the "flow protocol" (figg. 3, 4, 5 and 6) [16]. It enables communication between entities (modules and satellites with their hubs) on various network supports, ranging from simple serial TTY lines to Unix-domain sockets, TCP, UDP, and WebSockets offered by the HTTP support [17].

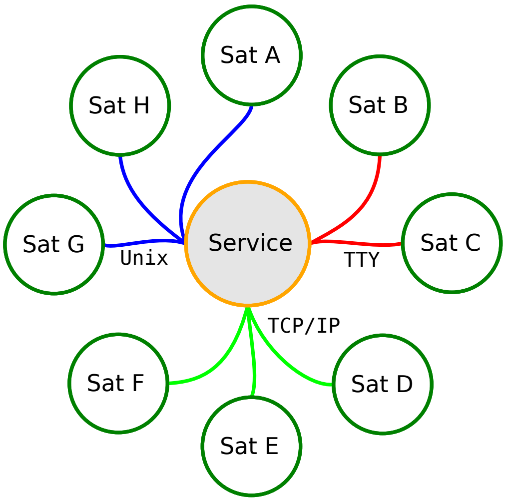

Figure 3: Mono-centralized FlowNetwork

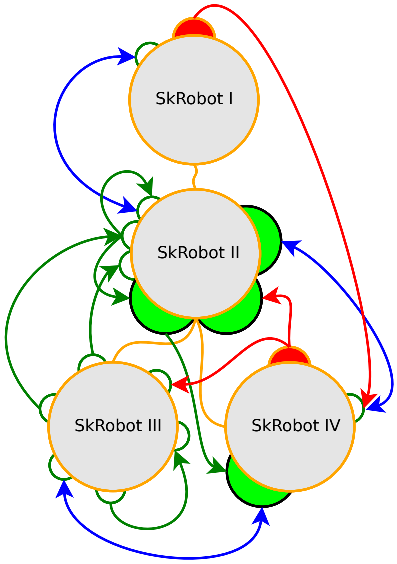

Figure 4: Multi-centralized FlowNetwork/DipoleNetwork

In this context, communication uses a binary format, transporting structured frames for each command and response. The protocol distinguishes between synchronous and asynchronous commands, which are crucial for service distribution and flow management. Synchronous commands block until a response is received, while asynchronous commands do not wait and may send notification frames to all or only relevant connections based on the request type. For

[16] https://gitlab.com/Tetsuo-tek/SpecialK/-/tree/master/LibSkCore/Core/System/Network/FlowNetwork last accessed June 2024

[17] https://gitlab.com/Tetsuo-tek/SpecialK/-/tree/master/LibSkCore/Core/System/Network/TCP/HTTP last accessed June 2024

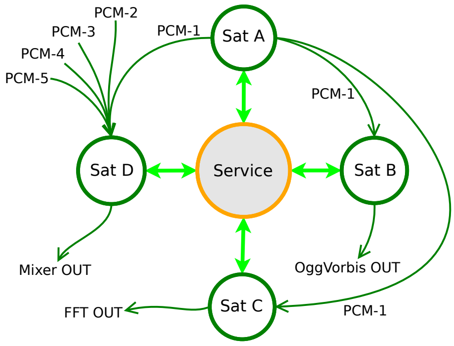

Figure 5: Example of audio distribution

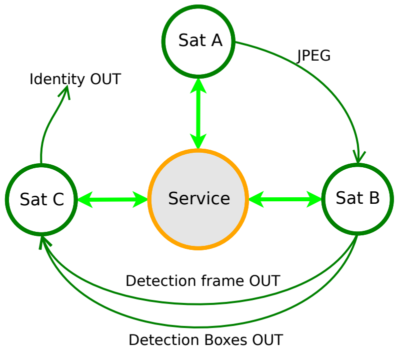

Figure 6: Example of video distribution

efficiency, the binary frame lacks control records for data quality and integrity, relying instead on the correctness of parameter order and the announced segment length to ensure frame integrity. Any communication errors result in terminating the connection, logged as "Killing spurious client". Due to the prevalence of little-endian (LE) hardware, all binary data within the frame is written in LE format, contrary to the big-endian (BE), called also Network-Order, typically used in other protocols. The *SkFlowServer* class manages these communications, supporting network functionalities and handling new connections, which can be either synchronous or asynchronous. Asynchronous connections facilitate the distribution of computational tasks across different processes and threads, aligning with the generic flow concept of the satellite.

Note that all connections accepted by the server always start as synchronous. Only after the authentication is passed is it possible to set the connection as asynchronous, transforming it into a satellite distribution flow. The commands in the synchronous section, some of which are also valid for asynchronous connections, consist of those related to authentication and database service management offered through the *SkFlowPairDatabase* class:

- login execution;
- database selecting creation, persistence saving, removal, heterogeneous data-retrieves and data management through database himself other than its pair variables.

All database operations are subordinated to the current database label setup before execution. The current database setup is executed every time the database target for performing operations is changed. Variables are stored in the form of *SkVariant* type, a class capable of containing and expressing all primitive types, buffer pointers, and other container types such as strings, objects, maps, vectors, and lists. *SkVariant* class can convert to/from Json its contents. Protocol commands that use variables are duplicated with a less efficient counterpart that transports textual JSON as an adaptation to porting platforms where the *SkVariant* is not available yet, such as Python. Other synchronous commands directed to channels allow the following:

- obtain the current list of channels and their properties (pair-variables) and data;
- register/deregister for polling on a streaming channel buffer;
- request the latest current buffer (polling) for a streaming channel;

- makes synchronous requests to an existing service channel and receives responses.

As mentioned before, asynchronous commands do not receive responses; at least they could trigger the generation of asynchronous messages to one or many targets based on the requested command:

- setting the connection as asynchronous (originally, it is always synchronous);
- request for creating a streaming or service channel;
- removing a channel of which one is the creator (owner);
- requesting subscribe/unsubscribe for an active channel;
- linking/unlinking a replicating channel to/from a source channel (attach/detach);
- requesting data publishing on owned channels;
- setting the current database;
- creating/removing a database;
- saving on disk for database persistence;
- setting/creating/removing pair-variables.

When obtaining pair variables and data is necessary, a temporary synchronous connection must be established to make the blocking request(s). The connection can be closed, and the object can be destroyed once the data has been obtained. In asynchronous connections, the messages received from the server refer to the following events (not related to Sk events described earlier but to the FlowNetwork):

- the current database has changed (only on the connection that set it);
- a channel has been created/removed (to all);
- a channel has modified its header (to all);
- a service request has arrived on a previously created service channel (to the owner);
- a (first/last) streaming-request to start/stop publishing data on a specific channel is happen (to the channel owner);
- streaming data has arrived from a channel that has been subscribed to (to all subscribers).

The distribution also occurs, primarily through flow channels that can be identified as data queues (1:N and 1:1) that overlap within the same asynchronous connection described earlier as a satellite flow. The transmission on flow channels always occurs asynchronously to avoid interference in the transit of different data types with various weights and speeds. A FlowNetwork can be activated up to 32768 flow channels (the type *SkFlowChanID* being a redefinition of *short*). All channels in a FlowNetwork, regardless of their type, are identified by three always unique values, gotten as tuple or individually: (chanID, hashID, name) . As mentioned earlier, channels can be of two types:

- 1:N - intended for the distribution of streaming data of any kind that can be subscribed to and thus received by multiple consumers,
- 1:1 - intended to offer a synchronous (or asynchronous) service based on request/response and with the possibility of creating one-to-one streaming due to service requests.

When the flow is disconnected, all embedded channels and relationships within the distributed network are removed and managed only by an administrator or owner. If the hub core owns the channel, its removal depends on commands from an administrator. The SkRobot architecture, which is modular and core-focused, manages communication via FlowNetwork. Internal modules enhance system communication and synchronisation, governed by the core from initiation to shutdown, deriving from the *SkAbstractModule* interface [18]. This setup manages internal parameters through a configurable JSON structure and requires a redefinition of virtual

[18] https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Modules/skabstractmodule.h last accessed June 2024

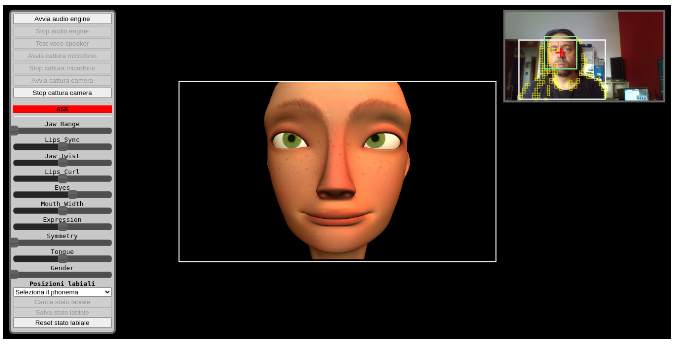

Figure 7: Robot agent web-user-interface screenshot

methods for transitions, control, and event handling. Developing internal modules is more complex than external satellites, for which the *SkFlowSat* class [19], similar to *SkAbstractModule*, automates connection, flow management, and event subscription. Both interfaces simplify network and event management, reducing repetitive coding, ensuring satellite code focuses on essential functionalities, and streamlining consistent, compatible component development within the network and service framework.

Examples of C++ and Python code can be acquired from SkRobot repository [20].

4. Results

SkRobot made it possible to create the robot-agent demo application [21]. It is a distributed and medium-complex satellite application that uses other secondary satellites to offer a virtual assistant service capable of listening, seeing, and operating procedures and commands on its system or other remote satellites (Fig. 7), inspired by our previous work [13]. The SkRobot application integrates core detection modules for faces, movements, and QR/Bar codes and collaborates with external satellites for speech-recognition services using local OpenAI Whisper technology [14]. Suppose an Nvidia GPU with adequate memory is available. In that case, the application leverages CUDA to enhance processing capabilities, significantly boosting the performance of Automatic Speech Recognition (ASR) and computer vision tasks using cv::cuda. The system has been successfully developed and tested on Nvidia's Jetson Orin AGX and Jetson Nano B01 platforms. Its user interface is web-based and compatible with major browsers like Chrome, Chromium, and Edge, although Firefox is excluded due to its lack of support for audio resampling. Upon accessing SkRobot through a web browser, the HTML, JavaScript, and CSS required for the interface are downloaded. The JavaScript initialises a WebSocket client that connects to the robot-agent application on the same IP, facilitating real-time interaction. Once the connection handshake is complete, the application displays a 3D avatar of the virtual assistant, based on the open-source "Armanda - 3D Talking Agent" by Patrizio Migliarini, PhD

[19] https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/System/Network/FlowNetwork/skflowsat.h last accessed June 2024
[20] https://gitlab.com/Tetsuo-tek/SkRobot/-/tree/main/examples last accessed June 2024
[21] https://gitlab.com/Tetsuo-tek/robot-agent last accessed June 2024

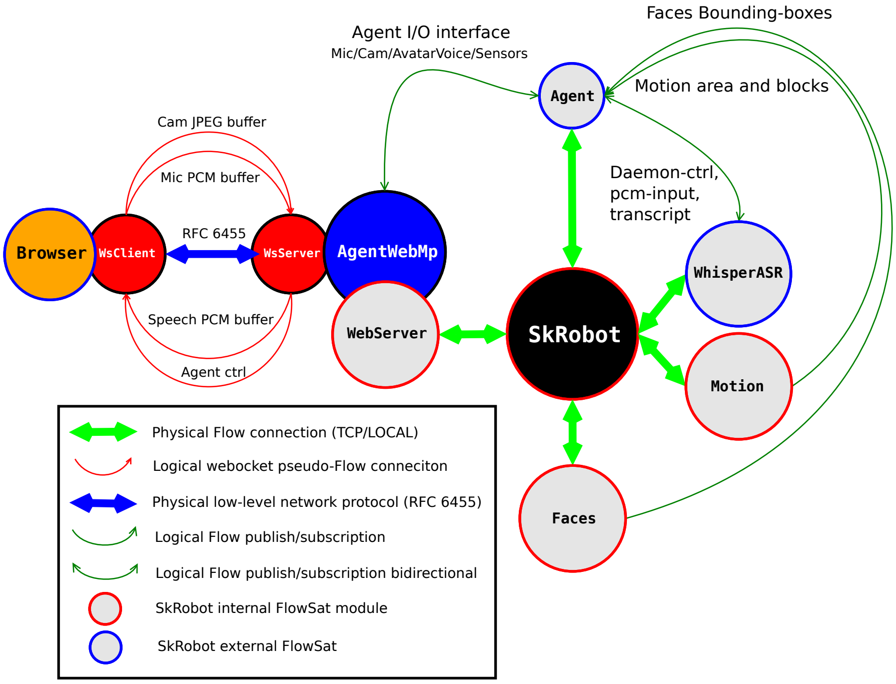

Figure 8: Robot agent FlowNetwork graph

[22] , which has been adapted and integrated into the SkRobot environment. The application graph is shown in Fig. 8. The SkRobot application allows users to activate their microphone and camera through the browser, using interface buttons and granting the required multimedia input permissions. Audio and video streams are captured and transmitted over WebSocket. The Audio Stream is sent as a binary sequence of 16-bit PCM buffers, mono, at a sampling rate of 22050 Hz. The Video Stream is sent as 640x360 resolution JPEG frames at 15 frames per second (MJPEG). The browser-based WebSocket client receives JSON text commands to control the avatar and associated actions, including eye blinking, gaze direction, and lip movement synchronised with synthetic speech from the Espeak library. Despite Espeak's simplistic and robotic vocal timbre, it supports Italian and operates efficiently on lower-performance devices. Binary PCM audio data representing the avatar's voice is transmitted back to the browser, with lip movements triggered by phonetic analysis performed by the Espeak engine, timed to match the speech. For network operations, in Video Processing, the original video frame is converted to a monochrome JPEG of just the Y channel (most significant for detection) for lighter processing. For Audio Processing, additional audio channels are created for the Fast Fourier Transform (FFT) of the microphone audio and ASR daemon control. Each collaborating satellite for speech recognition, named robot-whisper-asr [23], manages two channels: Audio Input for downloading audio from the microphone and transcription output for uploading transcribed text. The transcription daemon subscribes to the *Agent.ctrl* control queue and

[22]https://github.com/PatrizioM/armanda Last accessed June 2024
[23]https://gitlab.com/Tetsuo-tek/robot-whisper-asr Last accessed June 2024

multiple daemons can operate concurrently to enhance responsiveness, even during active transcription phases. Detector modules process and emit detection data (bounding geometries) through various queues. This data is aggregated and sent over WebSocket to the browser, where JavaScript visualises detected geometries overlaid on the camera preview. Facial expressions are implemented to enhance realism so that the avatar's blinking rate varies from 450 ms to 4 seconds, with distinct speeds for each blink phase. Also, Gaze Direction is controlled via data from face detection or the centre of motion if no faces are detected. This architecture allows for scalable parallel processing for ASR and detection services, optimising the system's efficiency and responsiveness. In this demonstration, the browser primarily acts as an executor, controlled by the application through pseudo-real-time remote commands. The only autonomous browser functions manage the FPS for camera capture and transitions during the blinking phase. SkRobot has also been utilised to retrofit the Nao v5 humanoid robot made by Aldebaran, revitalising its use in the Intelligent Systems and Robotics Laboratory (ISRLAB) at the University of L'Aquila. The Nao's original software, which is no longer updated due to deprecation, posed limitations on its use in educational activities. The Nao robot, programmable in C++ and Python-2, has faced challenges due to changes in ownership and lack of support, making its proprietary SDK inaccessible for modifications. Its Linux-based operating system is immutable, lacks a package manager or development tools, and complicates updates and software installations. Python-2 presents integration challenges with modern libraries and environments like ROS or Redis. Setting up development environments on external machines, especially on modern or Apple ARM-based systems, is problematic due to outdated dependencies. The flow protocol was ported to Python-2 as PySketch-py2 to bridge these gaps, enabling Nao to integrate with SkRobot. This adaptation allows satellites to connect to SkRobot, access data from Nao via its SDK, and transmit it across the flow network for processing by more advanced tools. Control commands can also be issued from satellites using Python-2, importing the Naoqi SDK directly [24].

Two Docker images have been created to support Python-2 development outside the Nao environment, containing PyNaoqi and SpecialK/SkRobot, respectively. Additionally, two PySketch examples, one for a publisher and one for a subscriber, facilitate communication between Nao and external applications outside the Naoqi SDK. This system architecture allows the distribution of sensory data and control commands across multiple satellites, supporting a scalable and distributed processing model for the Nao robot (Fig. 1). The satellite running Python-2 with PyNaoqi (NaoSat) also creates a service channel where all collaborating satellites authorised can send Json packages of control commands related to specific internal or external actions of the robot control can occur differently, with ad-hoc streaming queues created by collaborating satellites and subscribed to by NaoSat, obtaining an asynchronous stream of Json command-packages.

To manage all software related to SpecialK, SkRobot and PySketch, a meta-package manager was built: **pck** [25]. It is a meta-package manager based on git repositories. It was fast developed to manage the numerous applications, experiments and examples built around SpecialK, SkRobot, and PySketch. It is a very young software developed from scratch for specific contingent needs, but it is already capable of performing the required operations, such as setting up its

[24] https://gitlab.com/Tetsuo-tek/robot-nao-io last accessed June 2024
[25] https://gitlab.com/Tetsuo-tek/pck last accessed June 2024

environment and searching, installing, and removing applications available in the configured Repositories. Currently, the only configured repository is InoxPacks [26], which is the default for the environment. However, as explained below, it is possible to add new ones with different software sources. Pck is entirely written in Python and supports both versions 2.x and 3.x. In the next future, pck need to rewrite in an OOP and more clean form.

5. Discussion

ROS (Robot Operating System) supports C++ and Python 3.x, enhancing flexibility for developers. However, the exclusion of deprecated Python 2.x poses challenges for maintaining legacy systems like Aldebaran's Nao. This limits support for older robotic systems that are still used for education and research. ROS requires developers to manage event handling, synchronisation, and distributed information organisation, which can lead to redundancy and necessitates creating reusable objects. ROS's Callback paradigm for event handling can be limiting, complicating linking various functional scopes to a single event and supporting synchronised or real-time scenarios. Despite these challenges, ROS excels in asynchronous and parallel processing, making it a robust framework for modern robotics. Implementing ROS in educational and research settings is challenging due to its complexity and invasiveness, requiring specific OS management skills. This is particularly difficult on non-standard or minimal systems not based on Debian or Ubuntu. ROS's rapid evolution and frequent updates, often lacking backward compatibility, further complicate maintaining older projects. This necessitates careful consideration in environments where stability and ease of use are critical.

The learning curve for ROS is steep, posing challenges for students, researchers, and developers due to its demanding and fast-paced development environment. Despite these challenges, ROS remains a pivotal software in setting standards for distributed agent systems across various sectors, including robotics and IoT. ROS and SkRobot share core functional concepts that facilitate the development of distributed systems. An abstracted comparison of their properties includes:

- Utilization of a communication protocol with synchronous and asynchronous elements.
- Default intervals or synchronisation sources to regulate computational timing.
- Creation of data queues in a 1:N relationship for data producers.
- Subscription model allowing consumers to receive data from queues.
- Capability to transport any data type in real-time or pseudo-real-time through binary formats via asynchronous communication.
- Runtime modification and distribution of component-specific variables and parameters.
- Support for request/response transactions and streaming distribution, with the service provider constantly engaging in asynchronous communication.
- Support for multiple programming languages like C++ and Python through specific libraries and modules.
- Open-source licensing of the frameworks used.
- Enhanced system efficiency, computability, control, and resilience, with clustering and mirroring capabilities for automatic substitution and computational augmentation across multiple CPUs and machines.

These attributes underline the advanced capabilities of both ROS and SkRobot in managing complex, distributed systems efficiently. As described in Table 1, ROS and SkRobot share foundational concepts in distributed system design, yet they were developed independently,

[26] https://gitlab.com/Tetsuo-tek/InoxPacks last accessed June 2024

with ROS being studied and understood later in SkRobot's development. This resulted in convergent evolution, driven by common challenges within their domains, yet they differ significantly in their development philosophies and ecosystem structures.

Aspect	ROS	SkRobot
System Components	Nodes as atomic entities, complex relationships	Nodes (external satellites/central hubs), subnodes (internal satellites/modules), DipoleNetwork for dual-layer management
Development Approach	Robotics-focused, lower-level operations	Abstract design, pulsating processing, broad applicability, efficient notification engine
Code and Interface Management	Focuses on robotics, requires managing relationships	Uses *SkAbstractFlowSat* to simplify satellite/module creation, supports Qt/C++ and Python (2/3)
System Architecture and Management	Many dependencies, invasive, non-centralized in ROS-2, complex network configuration	Minimal OS changes, binaries/config files for workflows, multi-centralized hubs managing asynchronous messaging
Event Handling	Callback paradigm, limited flexibility	Signal/Slot paradigm, flexible event management, stable environment
Learning Curve	Steep, complex setup, frequent updates	Gradual, fewer dependencies, streamlined setup

Table 1
Comparative Analysis of ROS and SkRobot

While both systems provide robust frameworks for developing distributed applications, SkRobot offers a more developer-friendly environment with its simplified setup, broader system compatibility, and more abstract design philosophy. In contrast, with its detailed but complex system requirements, ROS provides a powerful but potentially cumbersome platform for specific robotic applications.

6. Threats to Validity

Potential biases and limitations could impact the findings of the comparative analysis between ROS and SkRobot. The primary concern is selection bias, as focusing exclusively on ROS and SkRobot may overlook other platforms providing different insights. The analysis relies on subjective interpretations based on individual experiences, which may not be universally applicable. The generalizability of the results is limited by specific use cases and platform dependencies, primarily Debian or Unix-like environments, potentially skewing the analysis towards these conditions. The rapid evolution of ROS and SkRobot introduces a dynamic factor, as frequent updates may quickly render some points outdated, affecting the long-term relevance of the analysis. Continuous development can lead to inconsistencies between versions, impacting stability and reliability. The analysis lacks empirical data to substantiate theoretical assessments, relying on inferred performance metrics. Variability in system configurations across implementations could lead to performance variations, complicating the application of the findings. While the analysis outlines functional differences and similarities between ROS and SkRobot, the interpretations are subject to the limitations of rapid development cycles, platform-specific dependencies, and broader application contexts. These factors should be considered to ensure the insights are relevant and appropriate for specific use cases.

7. Acknowledgments

This research was partially funded by the European Union - **NextGenerationEU** under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grant ECS00000041 VITALITY CUP E13C22001060006.

References

[1] D. L. Poole, A. K. Mackworth, Artificial Intelligence: foundations of computational agents, Cambridge University Press, 2010.
[2] S. Costantini, G. De Gasperis, G. Nazzicone, DALI for cognitive robotics: Principles and prototype implementation, in: Practical Aspects of Declarative Languages: 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings 19, Springer, 2017, pp. 152–162.
[3] R. Chrisley, Embodied artificial intelligence, Artificial intelligence 149 (2003) 131–150.
[4] W. D. D. Lawrence G. Mitchell, John A. Mutchmor, Zoologia, Zanichelli, 1991.
[5] P. H. Raven, R. F. Evert, S. E. Eichhorn, Biologia delle piante, Zanichelli, 1991.
[6] C. Moulin-Frier, T. Fischer, M. Petit, G. Pointeau, J.-Y. Puigbo, U. Pattacini, S. C. Low, D. Camilleri, P. Nguyen, M. Hoffmann, et al., Dac-h3: A proactive robot cognitive architecture to acquire and express knowledge about the world and the self, IEEE Transactions on Cognitive and Developmental Systems 10 (2017) 1005–1022.
[7] S. Costantini, G. De Gasperis, L. Lauretis, et al., An application of declarative languages in distributed architectures: ASP and DALI microservices., International Journal of Interactive Multimedia and Artificial Intelligence 6 (2021) 66–79.
[8] A. Dyoub, G. De Gasperis, Rule-based supervisor and checker of deep learning perception modules in cognitive robotics., in: RuleML+ RR (Supplement), 2017.
[9] D. Di Ottavio, Skrobot application server, an hub for flow-sat clients based on flow-protocol, 2024. URL: https://gitlab.com/Tetsuo-tek/SkRobot.
[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Ng, ROS: an open-source robot operating system, ICRA Workshop on Open Source Software 3 (2009).
[11] M. Dalheimer, Programming with QT: Writing portable GUI applications on Unix and Win32, " O'Reilly Media, Inc.", 2002.
[12] S. Monk, M. McCabe, Programming Arduino: getting started with sketches, volume 176, McGraw-Hill Education New York, 2016.
[13] S. Costantini, G. De Gasperis, P. Migliarini, Multi-agent system engineering for emphatic human-robot interaction, in: 2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), 2019, pp. 36–42. doi:10.1109/AIKE.2019.00015.
[14] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, I. Sutskever, Robust speech recognition via large-scale weak supervision, in: International conference on machine learning, PMLR, 2023, pp. 28492–28518.

Authorship Declaration As non-English native speakers, the authors occasionally used ChatGPT [27] and Grammarly [28] tools to improve the readability of the text. After using the tools, the authors reviewed and edited the content as needed and took full content authorship responsibility.

[27] https://chat.openai.com last accessed June 2024
[28] https://grammarly.com last accessed June 2024