CCFE
CULHAM CENTRE FOR FUSION ENERGY

UK Atomic Energy Authority

UKAEA-CCFE-PR(20)133

Robert Skilton, Guy Burroughes, Matt Goodliffe

Future-Proofing Robotic Maintenance and Inspection Systems by Abstraction and Standardisation of Distributed Communications

Future-Proofing Robotic Maintenance and Inspection Systems by Abstraction and Standardisation of Distributed Communications

Robert Skilton, Guy Burroughes, Matt Goodliffe

This is a preprint of a paper submitted for publication in
Robotics and Autonomous Systems

Future-Proofing Robotic Maintenance and Inspection Systems by Abstraction and Standardisation of Distributed Communications

Robert Skilton[a,*], Guy Burroughes[a], Matt Goodliffe[a]

[a] *RACE, UK Atomic Energy Authority, Culham Science Centre, Oxfordshire, UK*

Abstract

Future experimental facilities, hazardous energy generation assets, and decommissioning facilities will require ever greater degrees of remote maintenance, which will be facilitated by robotic systems. However, as the plant scales so does the complexity of the handling systems. Moreover, these long-lived (30+ years) handling systems will need to maintained and upgraded. Currently, there exist remote handling architectures are to highly coupled monolithic systems that evolve in their life-time in to unmanageable complexity. We propose an architecture, enabled by a novel communication protocol; which enable scalable standardisation, and novel distribution method that is suitable for mixed real-time applications of realistic network infrastructure. The architecture has been demonstrated in a laboratory environment, will be further deployed in real-world applications, and is directly applicable to many other robotics scenarios.

Keywords: Robot Communications, Robot Operating Systems, Software Framework, Networked Robotics, Distributed Control

*Corresponding author

Email addresses: robert.skilton@ukaea.uk (Robert Skilton), guy.burroughes@ukaea.uk (Guy Burroughes), matt.goodliffe@ukaea.uk (Matt Goodliffe)

URL: www.race.ukaea.uk (Robert Skilton), www.race.ukaea.uk (Guy Burroughes), www.race.ukaea.uk (Matt Goodliffe)

1. Introduction

In hazardous environments such as nuclear reactors where it is not possible or desirable to allow human entry, robotic systems are often employed to carry out necessary tasks such as inspection and maintenance. For large scale experimental devices like the ITER nuclear fusion experimental device [1] currently being built in the south of France, reconfigurability is paramount. In order to maximise their science value per dollar it is inevitably required that the facilities have a high level of flexibility so that they can track the science that they advance. However, as these devices have a natural tendency to become larger, more complex, and more hazardous, they inescapably require increased levels of remote maintenance. Maintaining these complex facilities is challenging due to having to deal with changing maintenance requirements, managing obsolescence in long-life systems as well as managing the impact of obsolescence mitigation activities, allowing for the integration of many items from different suppliers and nationalities.

We propose several mechanisms for addressing these problems, including a novel interface representation, communications protocol, and software framework. We take inspiration from the principles of Object Oriented Design (OOD), and attempt to solve the described problems using the principles of encapsulation and standardised interfaces. We demonstrate an approach that extends the principles of traditional middleware systems towards solving larger issues than just communications and remote procedure calls.

Initial results have proven highly successful and we present a number of application test cases. Although this work is motivated by nuclear fusion remote maintenance, the outcome is much more widely applicable.

Figure 1: A remotely operated robotic facility, designed to maintain the Joint European Torus (JET) nuclear fusion experimental reactor.

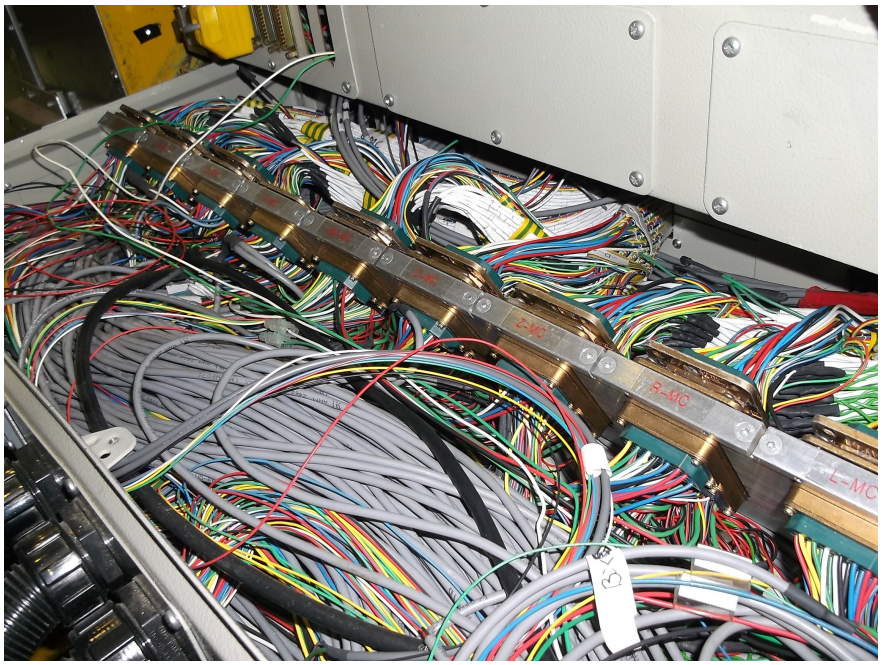

Figure 2: Monolithic architectures with high coupling and low cohesiveness often result in challenging scenarios arising from system evolution. This is often most clearly visible in system wiring.

2. Background

High energy physics research devices are often large, complex, and present environments which are hazardous to humans for reasons of elevated temperature, radiation, and toxic materials. Examples include the Joint European Torus (JET) experimental fusion reactor [2], the ITER experiment, The European Spallation Source (ESS)[3] currently under construction in Sweden, and the Chinese Fusion Engineering Test Reactor (CFETR) [4].

Due to the long lifespan of these experimental devices, ordinarily in excess of 30 years [], coupled with the changing requirements that results from the experimental nature of the facilities and accelerating external developments, maintenance and reconfiguration of equipment within these hazardous environments is frequently required. This is typically conducted using tele-robotic devices, and remotely controlled tools and equipment [5].

As such, the remote maintenance systems themselves can frequently require reconfiguration to meet the new needs of the plant being operated upon. Traditionally, architectures for remote maintenance have been focussed on delivery on short-term programme renewal schedule (i.e. 2 years). This inevitably pushes future-proofing to a low priority and results in ad-hoc, highly coupled monolithic architectures with low cohesion, reulting in degradation of quality of systems as shown in Figure 2. However, this reconfiguration scheme requires significant effort to be expended in order to make relatively small changes, and does not

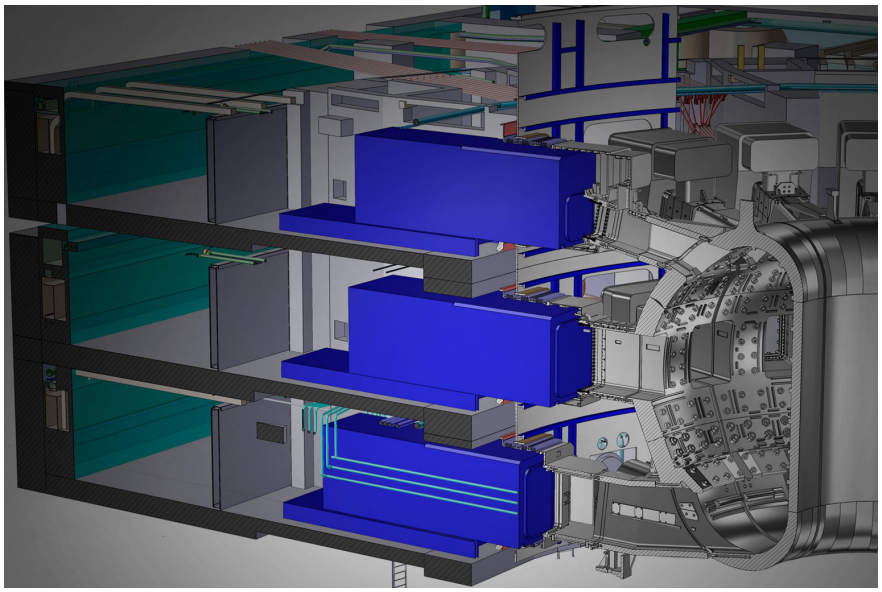

Figure 3: The ITER cask transportation system.

scale linearly with the scale of the system.

ITER and its supporting equipment will be vast, filled hundreds of complex devices all requiring advanced/experimental control systems developed by over 20 countries over a decade. For example, the cask transport system which will transport sealed cargo containers to and from the reactor and the advanced de-manufacturing hall. It will achieve this on an automated omni-directional vehicle that load and unload the cask safely. In a single cycle a cask transporter will interact with hundreds of different control systems, accessing controlled areas, interfacing and docking with airlocks, stowing and deploying various robotics components. This is not a unique example in ITER, almost all components will need to interface with hundreds of other in the same way. If JET is an indicator it will be maintained and upgraded for 30 years. Thus, ITER must be future-proof to advancements in technologies.

Also consider the human interfaces within a complex facility comprised of hundreda of robotic systems. How will maintenance staff support and diagnose issues in this system. How will operators control hundreds of separate control systems? Ideally, a single cohesive platform for human interaction could be generated allowing for a single point of access.

Another issue relates to the integration of many items from different suppliers, possibly different collaborating nations, into a functioning integrated capability. All of these systems will need to be integrated into a working capability, with generic human-machine interfaces, and will likely require a non-specialised operator workforce due to the expense of having individual teams trained on specific equipment.

All of these are essentially change management issues, and the problem can be phrased: “How can we allow for the ongoing, seamless removal, addition, or modification of any given subsystem within a remote

maintenance system whilst minimising impact on other subsystems?" The ideal scenario is one where a change to one subsystem does not require any change whatever in any other subsystem, yet all of the subsystems are able to continue functioning and interacting throughout the change process. This is also true for the human elements of the remote maintenance; the operators and maintainers should be minimally impacted by subsystem changes, ideally not needing to undergo time consuming and costly retraining. One further issue that presents itself when we try to standardise interfaces is that of bespoke vs. COTS systems. When a bespoke system is developed for a particular application, it is relatively simple to conform to a set of standard interfaces.

However, due to the need to reduce cost and take advantage of the reliability benefits of COTS equipment, bespoke systems are rarely developed (and shouldn't be) where COTS equivalents are available. We are therefore faced with a problem of providing standardised interfaces which can both be built into bespoke equipment, and added on to COTS equipment.

3. Related Work

Decoupling of systems and standardised communications are tasks traditionally performed using middleware platforms, so it is instructive to investigate existing middleware. Middleware can help software developers avoid having to write application programming interfaces (API), bespoke bus protocols, or highly tailored TCP message protocol for every control system, by serving as an independent communications interface management system for their applications. For operation over unreliable communications networks, use of mediator tools (middleware) can be of significant benefit since they generally provide built-in mechanisms and tools for management and supervision of Quality of Service and analysis of eventual failures in telecommunication services. Finally, middleware should assist in handling rapid and secure transactions over many different types of computer environments, providing independence from operating systems, programming languages, and communications transportation mechanisms.

Middleware and communication frameworks such as ICE [6], DDS [7], and Protocol Buffers [8] are mainly focused on providing communications functionality and so are not concerned with allowing flexible, decoupled evolution of systems.

This is also true for middleware designed for use with physical devices such as robotics; for example, ROS [9]. ROS is an open-source robot operating system provided by the Open Source Robotics Foundation. ROS provides a structured communications middleware layer above the host operating systems of a heterogeneous compute cluster. It was design with a philosophy of modular, tools-based software development, and is hence focused primarily on provision of reusable tools. It aims at maximizing the reusability of available robot sensor visualizations, sensor fusion and control algorithms. It has gradually managed to gather a large development community and it has become a popular framework for robotic platforms, primarily within

academic research.

ROS defines standard message types for commonly used robot sensor data such as images, inertial measurements, GPS, odometry etc. for communicating between nodes, thus separate data structures do not need to be explicitly defined for integrating different components. However, these messages have been created on demand and they are continuously evolving as new needs are identified, causing compatibility issues. ROS is a powerful communications tool, and ROS 2.0 will probably be a powerful communications tool capable of coping with real-time communications. However, it could never fairly be described as plug and play. Its overly permissive nature inevitably leads to highly coupled monolithic solution arising, causing lasting maintainability and upgradability issues.

ROS always requires a large engineering effort to integrate every device, no two systems are the same (e.g. USB cameras, IP cameras), ROS is a communications framework not a "systems of systems framework. Adding a new object into the systems, like an improved robot, is almost guaranteed to require an extensive amount of expert knowledge and engineering.

Finally, the lack of defined framework standards leads to reproduction of work and inconsistent designs naturally arise, (1 good standard is better than 50 great standards). This is especially true for user interfaces.

MOOS [10][11] suffers from the exact same problems as ROS, without the promise of real-time and determinism of ROS 2.0. MOOS does offer the MOOSDB, which acts as a central server for all communicated information in the system.

Another ROS-like system is EPICS [12], The Experimental Physics and Industrial Control System, is a software environment used to develop and implement distributed control systems to operate devices such as particle accelerators, telescopes and other large experiments. As with ROS, EPICS uses client/server and publish/subscribe techniques to communicate between the various computers.

GenRobot, based on Generis [13] is generic low level control system software controller for the RH Control System. It aims to solve the problems of control robots in uniform, reliable, and SIL2 fashion, which is crucial. But is not the same problem as posed in this paper. Similarly, OROCOS [14] falls into the same field of solving the problem of a deterministic middleware for robotics but other than this valuable distinction suffers from the same issues as ROS.

Another interesting example is TAO2000 [15], which is a core software platform dedicated to computer-aided force-feedback teleoperation. TAO2000 has been developed by the CEA LIST as a generic master–slave force feedback teleoperation system able to control different types of manipulators. Similar to many other middleware, most of the TAO2000's reusable components. However, unlike ROS and Orocos; TAO2000 is a tele-robotics oriented system, with a well-defined set of features, including control and graphical user interface (GUI) capabilities. Although some robotics modes are available and are frequently used during tasks, TAO2000 is dedicated to force-feedback tele-manipulation. It allows high-speed synchronization between

several real and virtual mechanisms (master arm, slave arm, camera, dynamic simulation engine, etc.). However, because of its specific nature its centralised system cannot aid in the larger problems of a system like ITER. But there are useful lessons to be learnt in it abstraction layers.

An alternative approach to use of this type of middleware framework is using a distributed industrial control systems frameworks, such as Another noteworthy system is SCADA [16]. Supervisory control and data acquisition (SCADA) is a control system architecture that uses computers, networked data communications and graphical user interfaces for high-level process supervisory management, but uses other peripheral devices such as programmable logic controllers and discrete PID controllers to interface to the process plant or machinery. Primarily, SCADA is monitoring and data logging systems which is supervisory software. It is not designed as a control processing framework leaving the underlying system to suffer from the issues described above. Similar to the previous mentioned middleware solutions, DCS and SCADA based systems require significant integration and reconfiguration effort. However, the fourth generation of SCADA demonstrates the utility of a distributed virtual model in monitoring and security concerns that occurs with systems with these characteristics.

Whilst our proposed system does not provide all of the features available in all of the described frameworks, it does address the more fundamental problem of impact propagation during system evolution, and therefore forward compatibility and future-proofing of complex systems of systems.

4. Philosophy of the Standardised Framework

In this section, we look at the basic goals and underlying principles of the solution, essentially forming a high-level description of the key considerations of such a system.

Distributed and Decentralised

To support a scalable network on robots and devices, it will be necessary for the framework support a distributed network of devices. Furthering this philosophy, there should be no central point in the distributed network as this would introduce a single point of failure and would also introduce a bottleneck to scalability.

Encapsulated

Implementation is entirely hidden, with only the standard, generic interface presented to the outside world. Modules should be self-contained, so that they can be verified and validated once and deployed in multiple situations. Modules should verify their input and output, so that the only object that control modules memory and functionality is the module itself.

Scalable Standardisation

Functions and data should be abstracted as far as reasonably practicable. To improve scalability, ensure maintainability, and future proofing large scale systems a degree of standardisation must take place. However, this must be tempered with the understanding that technology is developing faster than any committee can hope to understand let alone control. Thus, the correct tool must be developed to enable scalable layers of abstraction to be developed, balancing constraint and permissiveness. This should be achieved that if a modules API is extended, modules unaware of the extension should not be impeded by the extension. This is addressed more in Section 5.

Forward and backward compatible

The communications and data model shall be compatible with a broad range of inconceivable future systems, and it should be possible to modify the representation whilst maintaining compatibility with older systems which should not require modification. The system must therefore be unassuming about how it will be used, or which other systems may need to work with it.

Homogeneous

All participants present the same interface and interact in a functionally identical way. They should interact in the same manner regardless of physical topology (i.e. physical distribution). The modules to enable distribution and support any topology and make it easier for developer to deploy should agnostic distribution. To Distribution agnostic the modules should be unaffected whether the modules are connected to are local or remote. However, this should not infringe on the performance benefits of having local modules.

No Impact Propagation

Changing (adding, removing, or modifying) any given subsystem should never require a change to be made in any other subsystem (though some may be desirable to achieve specific interfaces). There should be no need to recompile any interfacing software when making a system change.

Middleware

The communications mechanism shall be independent of implementation including underlying operating systems and transport mechanisms, thereby not becoming an integration or obsolescence risk itself.

Self-Describing units

A system shall provide not only its data, but also a definition of the structure and meaning of that data, such that other systems can interpret and use the information. This goes towards developing a transparent connection, but a unique and usable API, whilst also adding additional semantic data. The communications

protocol should be self-describing in a manner that doesn't require expert level knowledge, or compilation of source code for two systems to communicate. Additionally, this information should include semantic information about the module that it is describing, to increase the possible functionality and improve discoverability.

Distributed database

A system should make no prior assumptions about what its data will be used for. It simply publishes everything. Other systems can make use of this information as they individually require. Everything should be made available. The middleware should act like a database, as single point of access. Data can be added, removed, updated, and queried all in an atomistic fashion. Whilst being decentralized and distributed.

Deterministic

The framework should support a repeating control cycle with a deterministic order of operation. In particular, the framework should minimise jitter and lag, whilst also behaving predictably when time constraints are violated. Jitter is the deviation from true periodicity of a periodic signal, such as reading or writing to and from a hardware device whilst lag is the absolute delay of propagation of that signal. Furthermore, whilst parts of these systems may contain the usual hard periodic tasks, requiring known bounded response time and controlled jitter, other parts may be made of much more complex soft (or non-hard) tasks to deal with sporadic or aperiodic events, leading to a considerable task diversity. Moreover, with distributed systems there is a requirement for a suitable communication network capable of dealing with the new demands task diversity dictates. Indeed, network protocols must deal with different traffic patterns and must provide not only controlled jitter and bounded message transmission time, as required by the usual hard tasks, but also high throughput, as demanded by soft and other non-periodic tasks. As a result, industrial communication networks, known by being reliable and predictable, may not suffice anymore due to their usual low bandwidth

The framework must be able to cope with poor, wireless, and realistic network infrastructure in a manner that will allow for control system to operate and communicate; although, perhaps with reduced functionality.

To start complex system in a low risk manner more than just its execution must be deterministic, it's necessary for a system to be initialised and de-initialised in a deterministic fashion. The framework should enable this even over a distributed network.

Dynamic

In a distributed modular system, the system should be capable of supporting dynamic insertion and removal of modules in the graph. Each of those modules should be capable of dynamically changing their configurations at any time. One or more monitoring system should monitor be able the state of the system

as a subgraph or full topology. The modules or an external system should be able to dynamically change the module's connections (the dataflow) to other connections. All of these capabilities together in a framework allows for complex systems which are capable of dynamic behaviour. The network of modules could become vast, to a point where a single computer could not understand it. A framework should enable a system to find and connect to only the modules it needs to, i.e. connecting only to a subgraph.

Generic interface tools

The architecture should allow for the implementation of generic interface tools which can be used to work with any possible compatible system.

5. A Standardised System Representation and Protocol

A communications protocol and framework has been developed to address the issues presented, and achieve the stated benefits. This consists of a data representation, and a transport protocol, to distribute the data, which varies depending on the underlying implementation.

The standardised data representation is in the form of a sparsely connected graph of generic nodes representing system elements. These nodes are known as Simplexes. Simplexes are the primary elements of the architecture. A simplex is a single unit of functionality, highly encapsulated, and highly cohesive. Every simplex is master of it domain, it's the only thing that can change its data and control its domain, thus preventing strong inter-simplex coupling.

5.1. Data Representation

The data is encoded in a specialised form which is well suited to representation of generic data and interfaces as it is allows for the definition of complex data structures, whilst ensuring high performance in a real-time context.

A 'model' of the data is constructed, which has a graphical structure. The complete system is represented as a graph of Simplexes which comprise the system at an appropriate level of granularity.

The model is transmitted using a publish-subscribe communications service, resulting in a novel distributed database model. Where all the data is placed into a small number of structured and defined tables in a distributed database. This enables a flat unchanging protocol that can universally accessed, whilst allowing query-able extensible data.

Some of tables are kept in a distributed p2p fashion. Other can be subscribed to by attaching to their respective clusters. Their type will be noted in the following descriptions.

The data is then further divided into a number of subsets as follows:

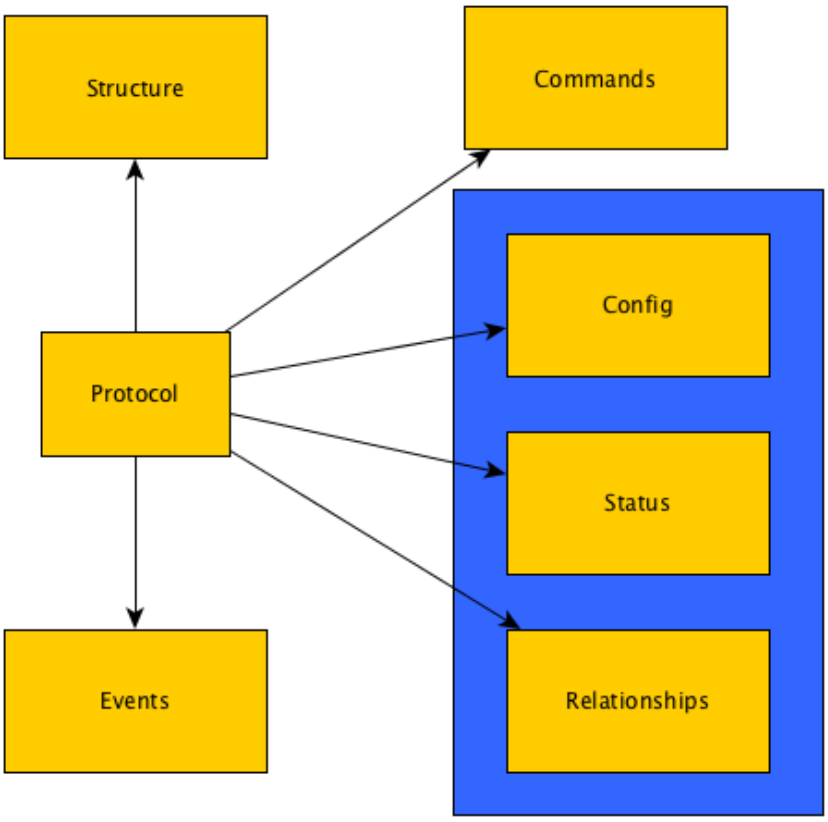

Figure 4: Key components of the protocol

5.2. Simplexes

A table of all the simplexes available on the network are kept in a distributed table. This table includes enough information for dynamic discovery of Simplexes. This information includes: Global Unique Identifier (GUID) of the simplex; Cluster in which the simplex is hosted; Domain and sub domain in which the simplex operates, an ID for the simplex in the sub-domain, simplex type hierarchy, and some semantic tags about the simplex.

5.3. Structure

The structure representation which defines the simplexes in a cluster, including all of a simplexes Data, Commands, and additional semantic information about the Simplex. Defining the structure of the memory for simplex and its cluster permanently. This allows for communications to occur without use of dynamic memory, which is necessary for Real-time capabilities.

5.4. State

The state of the Simplex consists of the full set of signals which are exposed by the Simplex for external use. Typically, this contains all of the variables that are used by or represent the condition of the node. The state is further divided into status data (information that is expected to change frequently – at or near the loop cycle frequency) and config data (information that changes infrequently or as a result of specific irregular events or commands).

Information here is stored in key-value sets, where the value may be a single value or an array of values. This simple mechanism allows a wide range of possible data representations such as scalar values, text strings, vectors, matrices, quaternions, etc. to be stored in an entirely generic way. This essentially forms a hybrid relational and key-value database representation of the system's data.

Critically, this generic representation means that all data is represented in a globally understood way, meaning that it is impossible for a Simplex to convey data that cannot be understood by any other Cortex system.

5.5. Control Interface

A control interface is represented by two main items: A definition of the commands that the node can carry out; an indication of the commands that are available or valid at the present moment; and a mechanism for the command to be used. All of which are in three separate tables.

5.6. Events

Events are discrete things that take place. Examples include equipment faults, changes of mode, completion of processes, etc. Events are published by Simplex only once, when they happen and kept in a global table. It is generally used to system-wide notifications, and monitor diagnostic information.

5.7. Transport Protocol

This Protocol can all be transmitted over a publish-subscribe mechanism such as DDS, a client-server mechanism such as TCP/IP, or a local mechanism such as shared memory or file storage. All communications are associated with QoS requirements, which includes timeliness and liveliness, as well information about when the internal data was generated. This makes it ideal for dealing with distributed real-time systems or systems on realistic and poor network infrastructure (e.g. Wi-Fi). It can also support various and variable frequencies.

P2P Persistence servers that exist as separate applications use a publish-subscribe model to enable a p2p distributed database each managing a set of lines in the table and updating remotes, ensuring their correctness. This mechanism is used for dynamic discovery primarily and not for high throughput or real-time necessary communications. Another feature of the Distributed Database method, is that enables the use of a query language to allow rapid, effective and structured filtering of the table. Most importantly, the benefit of the distributed database model is that it guarantees continuous real-time availability of all critical information.

6. A Standardised Framework

The architecture is comprised of a software layer, which embeds the standardised representation of the data and interfaces in the form of a sparsely connected graph of highly encapsulated Simplex nodes.

Simplexes are the primary elements of the architecture. A simplex is a single unit of functionality, highly encapsulated, and highly cohesive. Every simplex is master of it domain, it's the only thing that can change its data and control its domain, thus preventing strong inter-simplex coupling.

Each Simplex is composed of 3 elements:

- The Data, which is the publicly exposed data.
- The Commands, which are akin to remote procedure calls, but are more like functionality requests.
- The Functions, which is the repository for the actual algorithmic functionality.

Simplex data and interfaces are represented in a standardised way, providing homogeneity, and enforcing abstraction.

A control system or similar is then constructed from a collection of Simplexes connected together. Reusable subsystems can be constructed from collections of connected Simplexes.

6.1. Data

Internal data is exposed to enable external systems to make use of it when it is may be useful to external parties. Examples of system types include: control systems, diagnostics and Graphical User Interfaces (GUIs).

Data is further sub-divided into Status, Config, and Relationships.

To respect real-time and network performance constraints Status data is limited to the data that is expected to change regularly, i.e. during normal operation, it is not unreasonable to expect this data to change between control or communications cycles. Configuration, or config data consists of the subset of data that is unlikely to change regulary. Each item of config and status data also has up to 3 levels of associated minimum and maximum limits: normal, abnormal, and exceptional.

Relationships are distribution friendly pointers to other data and simplexes. Relationship data is akin to Configuration data in its non-periodic nature. Relationships also define the types of dependency a simplex has on a relationship, which can be build-time, run-time, or no-dependency. On top of this relationships can have three type: necessary, which defines a relationship for which the simplex cannot operate without; preferable, which defines a relationship for which the simplex can operate without, but with a reduced functionality; and optional, which defines a relationship which is entirely optional. These relationships can then be used to form multiple directed acyclic graphs (DAGs).

These DAGs enable determinism whilst remaining encapsulated and distributed. For example, the DAG of build-time dependency relationships determine the necessary deterministic start-up routine, whilst the run-time dependencies DAG determines the deterministic execution order. With the addition of Quality of Service (QoS) information about data generation timestamps determinism can even brought to poor and unreliable networks or even merge of hard and soft systems.

These DAGs can also be used to generate meta-state-machines, composed of the state machines of numerous simplexes, and automatically progress them safely.

6.2. Commands

Commands are requests for functionality in a Simplex, and are conducted asynchronously and without response. Command sets can be seen as exposing functionality to the outside world, and are all of a standard form. Examples of commands might include changing from an initialised state to active state in an internal state diagram, or a progressing a robotic arm to a commanded waypoint. The commands can be interpreted and acted upon as the Commanded Simplex wishes, in keeping with the simplexes being encapsulated. This Command mechanism has been designed to ensure that all data is uniformly exposed, and that the standard operation of the simplexes are always the only focus of the simplexes, as to be more deterministic and distribution agnostic.

6.3. Functions

The final element is the Functions, the actual algorithmic functions it is how the simplex operates. The separation exists to improve the distribution agnosticism as it forms a distinct separation of the "API" of a simplex (the Data and Commands) and its internal workings (the Functions). One of the benefits of such as

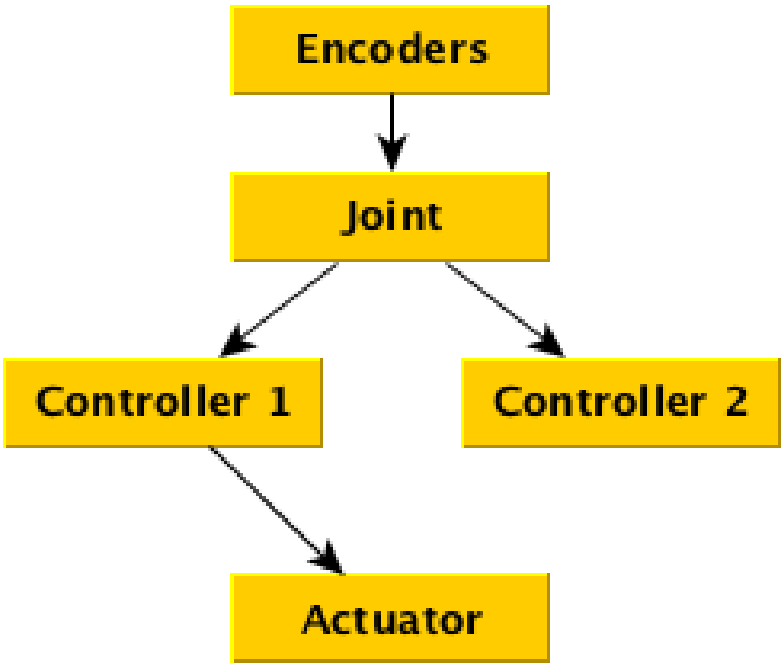

Figure 5: Illustrative example of Simplexes controlling a motor. The yellow boxes represent Simplexes, and the arrows represent dataflow.

separation is that the Functions can be easily replaced and other simplexes would be unable to distinguish them. This is particularly useful for simulating Simplexes, or extending their functionality.

6.4. Pulling in Data, changing relationships and commanding other Simplexes.

Then with these 3 elements all simplexes operate reading other simplexes data, based on their relationships, which can be changed at runtime. Performing their appropriate functions at the appropriate deterministic rate. Whilst responding to commands as they see fit.

When a Simplex reads the Data from another simplex it is only simplex that has the correct domain knowledge on what it requires. For these reasons, the initiating simplex will interpret the contents of the read Simplex, by attempting to mask the contents. This will be achieved by checking that correct elements in the simplex database are fulfilled, whilst ignoring the extra components. This enables simplexes to easily extend standards adding extra data elements or commands without effecting reliant simplexes. Additionally, with the use of relationships, traditionally incompatible standards changes can be mitigated via simply creating a mask to data from relationships.

For example, consider the control of a motor which can be performed a Simplex arrangement as in Figure 5. The initial sensing is performed by the Encoder Simplex, which then places its data in its status data.

This is then read by a Joint simplex based on a relationship from the Joint Simplex to the Encoder Simplex, and is translated into joint information that is placed in status data. Then, if both Controller simplexes are active, they both read the Joint information, and produce demand information. The Actuator Simplex then based on its controller relationship read the demand from Controller 1 and implements it. This alternatively could change based on the Actuator changing it relationship based on Controller 2 having a better QoS or some attractive Config value. Alternatively, an external source could have commanded the actuator to now listen to Controller 2. The Actuator does not need to know what the controller is, only that it includes the required data (which can be referenced by ID or Relationship) and commands, which may be a subset of the full command set. The actuator can then interpret as it wants. This allows for minimal impact extension.

Having the API in this flat fashion means that regardless to its extension the simplexes depending upon it can still operate as if it was not extended.

6.5. Distribution

This architecture not only allows for the flexible interoperation of control systems within a remote maintenance facility, but also facilitates the possibility of a fully distributed control system. No extension to the architecture or communications is required. Nodes simply exchange data which is already being broadcast, forming a control scheme which can be distributed across processes, across machines, or even the internet.

Distribution is achieved by grouping simplexes into Clusters. The Clusters then communicate with each other as necessary using the CorteX protocol (Section 7). The clusters communicate the CorteX protocol as a bus, communicating all of the Simplexes inside the cluster as one. This obviously helps constrains complexity and minimises communication overheads.

From the point of view, of a cluster communicating with another one it clones in the Simplexes from that cluster minus the Functions, as illustrated in Figure 7. These cloned Simplexes can then be asynchronously kept up to date with their original counterparts. From the point of view of the simplexes in the cluster, the clones are indistinguishable from their local cousins, exposing the exact same interface.

It is interesting to note at this point, that once cloned, alternative passive Functions can be put in the cloned Simplex. This can be used to achieve Graphical User Interfaces or Virtual Reality representation of physical object the Simplexes measure

7. Other Considerations

With personal data and IP stored on connected devices as well as devices potentially capable of causing injury to a person, cyber security is a necessary enabler of this technology, and if it is not prioritised the business opportunity will be undermined. And simply encrypting communications is not enough, a considered approach to an entire system's design is necessary. Thus, the simplexes have been designed to be suspicious

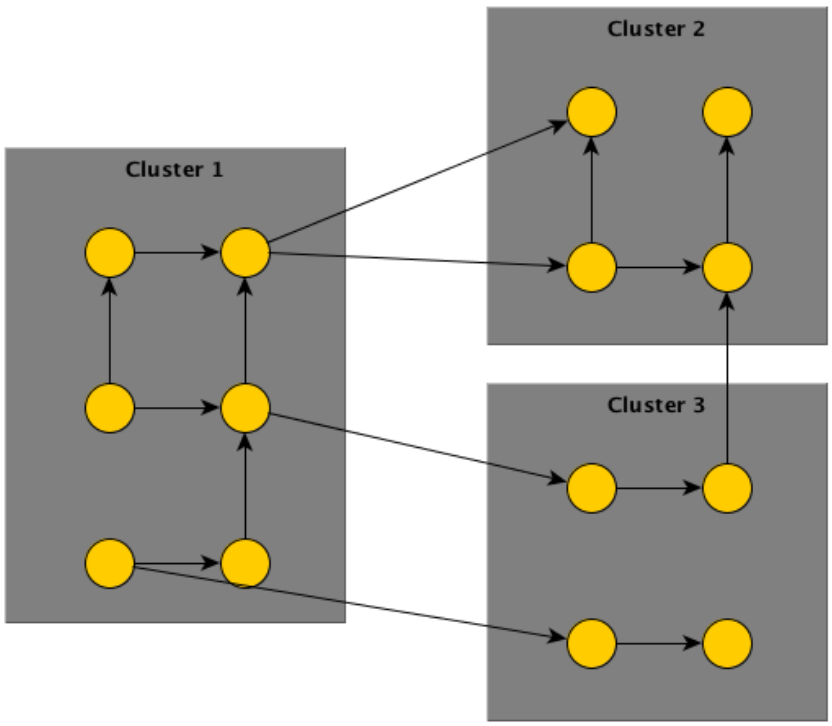

Figure 6: Simplexes in clusters

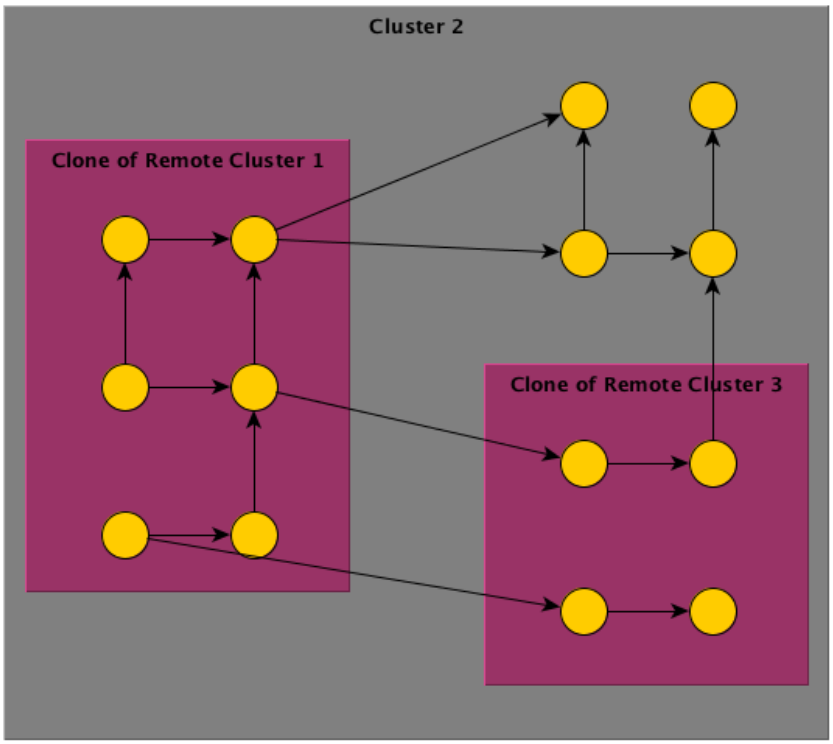

Figure 7: Cluster 2 from Figure 6 with it clones of remote simplexes.

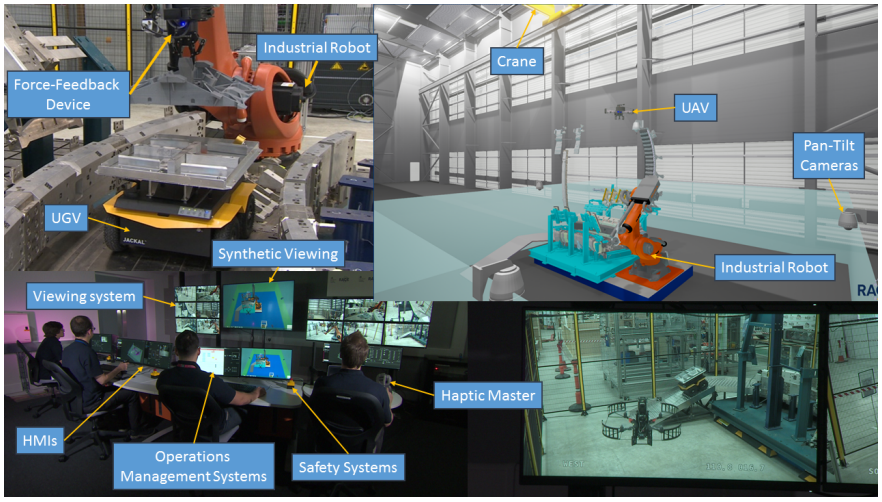

Figure 8: Lab trial elements

and self-contained, meaning that all inputs are interrogated as if it was malicious code. This distrust is one of the reasons that there are no request response style calls, as no response would ever be trusted without checking the data coming the simplex, thus making useless. Additionally, user-defined simplexes are run as plugin on top of a fault-tolerant, security aware clusters, which adds a layer of abstraction to improve security. Cyber-security is an endless topic; however, this platform should enable a reasonable cyber-security routine to be implemented. Additionally, Simplexes are self-contained, so that they can be verified and validated once and deployed in multiple situations. Simplexes should verify their input and output, so that the only object that control Simplexes memory and functionality is the Simplex itself.

One of the benefits a platform that has its data separated from its functionality is that an interpretation layer can be implemented to allow a plugin system to easily change the runtime implementation of simplexes; i.e. from real to simulated functionality. Additionally, this plugin system allows for closed source simplexes to be easily implemented.

8. Lab Trials

Initial trials have been conducted at the RACE testing facility. In order to validate the capability of this architecture within the context of a remotely operated robotic facility, a complete end-to-end technology demonstrator was constructed. This contained all major subsystems that would be found within a typical operational tele-robotics facility including viewing systems, emergency stop systems, planning tools, and human interface devices.

A selection of robotic devices was chosen in order to maximise the variety of devices demonstrated in order to demonstrate compatibility with a wide range of systems. This included an Industrial manipulator robot, mobile ground robot, aerial quadcopter robot, gantry crane, and a haptic interface device.

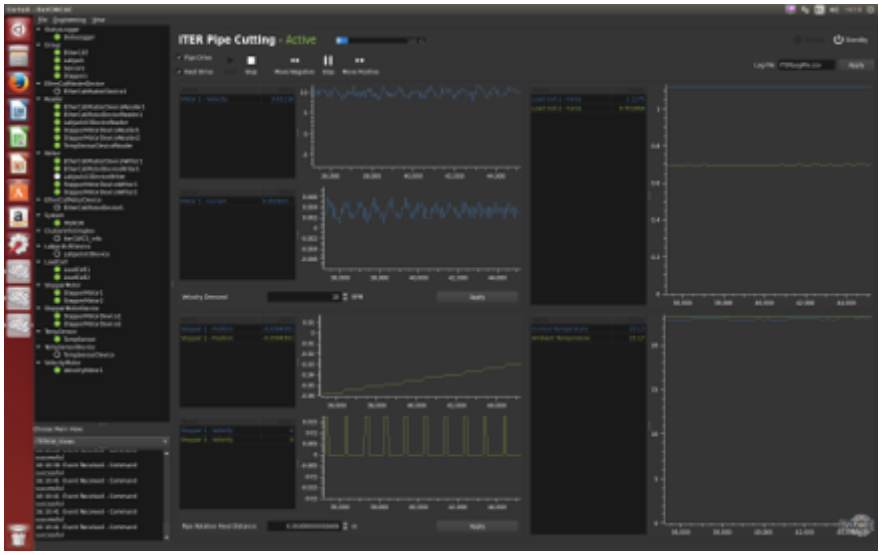

Figure 9: Auto-generated C&C HMI

8.1. C&C HMI

A generic Command and Control (C&C) Graphical User interface has been constructed that is designed to be compatible with any possible CorteX system. In some ways, this element is the most critical to both de-duplication of subsystems within a facility, as well as minimising operator training and retraining requirements. The GUI is multi-platform and features a plugin system to allow for customisation without the need to recompile or re-validate.

8.2. Virtual Reality Visualisation

Gateway wrappers to several visualisation tools, such as VR, synthetic viewing, and physics simulations, which can easily be created by implementing virtual reality functions for virtual simplexes cloned from remote simplexes.

8.3. Integration Times

The integration time control system and GUI development for a single engineer for these devices as listed in Table 1. These values are created from the point of view of the engineer having knowledge of devices, but not the particular model and makes of devices.

The rapid development time was achieved using existing Simplexes, their scalable Standardisation enabling use of their functionality with never before seen proprietary COTS.

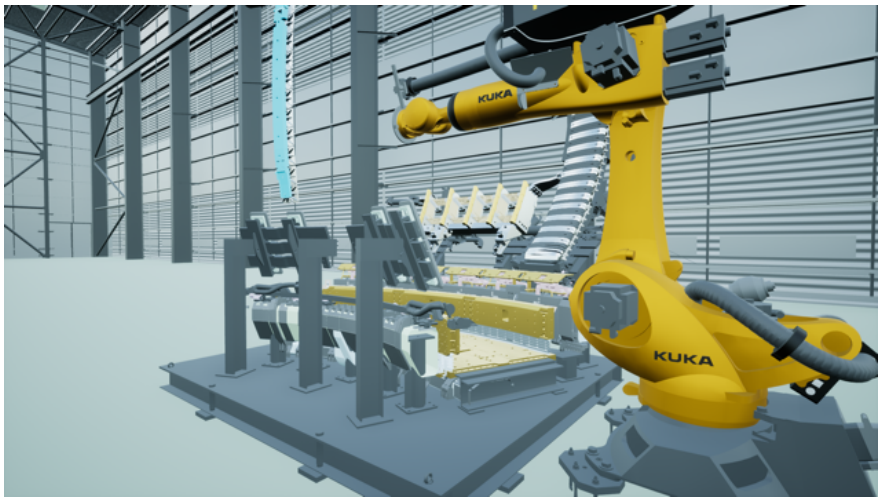

Figure 10: Real-time VR synthetic camera view of the experimental setup.

Table 1: Integration times for each of the main devices in the trial. Times are measured from having the system operational under its own control system, to full integration with the CorteX architecture.

Item	**Integration Time**
Industrial Robot (Kuka)	3 days
Haptic Master (Phantom)	2 days
UAV (Pelican)	2 days
UGV (Clearpath Jackal)	2 days
Crane Tracking (Vision-based)	2 days
Viewing System	3 days

9. Conclusions

CorteX demonstrated its ability to rapidly develop complex control systems that have all benefits discussed. The 4 main significant novel elements of CorteX are:

A communications protocol that allows collections of devices and software to dynamically connect and collaborate without a priori knowledge of other systems, by representation of a system as a graph of generic nodes with self-describing data.

A software framework that allows systems to be integrated, and new tools to be developed with inherent forward compatibility and scalable standardisation.

Virtual cloning the remote simplexes to make the distribution in the system seamless.

The use of data-flow (Relationships) to generate build-time and run-time directed acyclic graphs for scheduling.

Other notable features are the consideration made for mixed real-time requirements, ability to cope with poor networks, and consideration to cyber-security. For ITER-like remote maintenance, CorteX would provide a decentralised, distributed network that had all the data securely accessible, and that allowed for easy extension and repair. Additionally, the encapsulated nature of the system would remove the need for duplication of elements, by simply reusing existing simplex in other contexts. This deduplication would reduce cost and reduce risk of failure and the de-duplicated functionality could be verified and validated to a greater degree with the recovered resources.

10. Future Work

Clearly, for deployment in ITER-like system there should be considerations and research carried out into how to fit to nuclear regulation in a development friendly manner. Can a framework put in place for high complex software (i.e. machine learning) in a manner where the framework takes most of the burden of compliance and ensuring reliability and predictability.

CorteX will need a stronger test case; i.e. evaluation within the context of remote maintenance facility. This will be achieved by deployment within the European Spallation Source (ESS) facility scheduled for first operation in 2019. ESS will be a facility that will need to be maintained for the next 50 years, with the expectation of remaining on the cutting edge of technology.

References

[1] Bart Verberck. Building the way to fusion energy. *Nature Physics*, 12(5):395–397, 2016.

[2] JET Team et al. Fusion energy production from a deuterium-tritium plasma in the jet tokamak. *Nuclear Fusion*, 32(2):187, 1992.

[3] C Darve, M Eshraqi, M Lindroos, D McGinnis, S Molloy, P Bosland, and S Bousson. The ess superconducting linear accelerator. *Proceedings of SRF2013*, 2013.

[4] Yun Tao Song, Song Tao Wu, Jian Gang Li, Bao Nian Wan, Yuan Xi Wan, Peng Fu, Min You Ye, Jin Xing Zheng, Kun Lu, Xianggao Gao, et al. Concept design of cfetr tokamak machine. *IEEE Transactions on Plasma Science*, 42(3):503–509, 2014.

[5] Rob Buckingham and Antony Loving. Remote-handling challenges in fusion research and beyond. *Nature Physics*, 12(5):391–393, 2016.

[6] Stefan Niemczyk, Stephan Opfer, Nugroho Fredivianus, and Kurt Geihs. Ice: Self-configuration of information processing in heterogeneous agent teams. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 417–423, New York, NY, USA, 2017. ACM.

[7] G. Pardo-Castellote. Omg data-distribution service: architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, pages 200–206, May 2003.

[8] Kenton Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*, 72, 2008.

[9] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[10] Paul Michael Newman. Moos-mission orientated operating suite. 2008.

[11] Michael R Benjamin, Henrik Schmidt, Paul M Newman, and John J Leonard. Nested autonomy for unmanned marine vehicles with moos-ivp. *Journal of Field Robotics*, 27(6):834–875, 2010.

[12] Stephen A Lewis. Overview of the experimental physics and industrial control system: Epics. *h ttp://csg. lbl. gov/EPICS/OverView. pdf*, 2000.

[13] Emilio Ruiz Morales. Generis: The ec-jrc generalised software control system for industrial robots. *Industrial Robot: An International Journal*, 26(1):26–32, 1999.

[14] Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

[15] P Gicquel, C Andriot, F Coulon-Lauture, Y Measson, and P Desbats. Tao 2000: A generic control architecture for advanced computer aided teleoperation systems. In *Proceedings of the 9th ANS topical meeting on robotics and remote systems*, 2001.

[16] Stuart A Boyer. *SCADA: supervisory control and data acquisition*. International Society of Automation, 2009.