frontiers in Robotics and AI

TECHNOLOGY REPORT
published: 25 April 2016
doi: 10.3389/frobt.2016.00020

NUClear: A Loosely Coupled Software Architecture for Humanoid Robot Systems

*Trent Houliston**, *Jake Fountain*, *Yuqing Lin*, *Alexandre Mendes*, *Mitchell Metcalfe*, *Josiah Walker and Stephan K. Chalup*

Newcastle Robotics Laboratory, School of Electrical Engineering and Computer Science, The University of Newcastle, Callaghan, NSW, Australia

This paper discusses the design and interface of NUClear, a new hybrid message-passing architecture for embodied humanoid robotics. NUClear is modular, has low latency, and promotes functional and expandable software design. It greatly reduces the latency for messages passed between modules as the message routes are established at compile time. It also reduces the number of functions that must be written using a system called co-messages, which aids in dealing with multiple simultaneous data. NUClear has primarily been evaluated on a humanoid robotic soccer platform and on a robotic boat platform. Evaluations show that NUClear requires fewer callbacks and cache variables over existing message-passing architectures. NUClear does have limitations when applying these techniques on multi-processed systems. It performs best in lower power systems where computational resources are limited. This article aims at readers with interest in modern software engineering concepts and development of systems in areas such as robotics, smart devices and virtual reality.

Keywords: humanoid robot, robot vision, robot learning, software architecture, message passing, blackboard, co-messages, compile time message routing

OPEN ACCESS

Edited by:
Lorenzo Natale,
Istituto Italiano di Tecnologia, Italy

Reviewed by:
Neil Thomas Dantam,
Rice University, USA
Torbjorn Semb Dahl,
Plymouth University, UK

****Correspondence:***
Trent Houliston
trent.houliston@newcastle.edu.au

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *01 November 2015*
Accepted: *29 March 2016*
Published: *25 April 2016*

Citation:
Houliston T, Fountain J, Lin Y, Mendes A, Metcalfe M, Walker J and Chalup SK (2016) NUClear: A Loosely Coupled Software Architecture for Humanoid Robot Systems. Front. Robot. AI 3:20. doi: 10.3389/frobt.2016.00020

1. INTRODUCTION

A system's software architecture is the arrangement of its high level structures based on the layout of its functional code elements. As the software components of modern humanoid robotic systems become more capable, their code becomes larger and more complex. This can increase the cost of maintaining and enhancing the system. Software architecture design improves high level structures encouraging better maintainability and re-usability of components, and supports the reduction in effort in understanding and modifying software systems. However, this is difficult in embodied humanoid robots where computational hardware is limited in power, as architectural decisions to improve the maintainability of the system impact the performance of the robot. Therefore, architectures for this domain should aim to improve the efficiency for systems with limited performance.

Latency is also a key concern in robotic systems. Information must flow quickly between components for real-time control. Latency between sensing and acting reduces the robot's ability to correct issues in time. Architectural decisions that increase modularity also increase the latency between components, as their interfaces become more general. This impacts on the robot's ability to process and function as required.

The NUClear framework has been designed to address these concerns using novel techniques to improve the communication between modules. It utilizes C++ template meta-programing and smarter interfaces to reduce or eliminate costs while maintaining an expressive and easy to use interface.

2. ROBOTIC SOFTWARE ARCHITECTURES

When designing or improving software architecture for autonomous robots, it is important to analyze the techniques of existing systems. The software architectures used in the majority of autonomous humanoid robots are within a spectrum between two primary architectural paradigms. These systems can be defined by the way in which data is communicated between modules in the system. On one end of the spectrum are global data store systems, in which all data are stored centrally and accessed by expert system modules. At the other end of the spectrum are message-passing systems, where systems receive messages and perform actions on the data received. Systems also exist within this spectrum that have a degree of hybridization (**Figure 1**). These hybrids utilize message passing and also include features from the global store. This section of the paper explores various properties that contribute to the architecture quality of existing systems. Also discussed are techniques available to improve the performance of the system.

2.1. Blackboard

A blackboard architecture is a form of global store architecture (**Figure 2**). Modules within a blackboard system communicate with each other through the manipulation of data elements stored on a central data store called the blackboard. This manipulation is achieved through expert systems (Hayes-Roth, 1985) that are responsible for performing particular tasks. For example, an expert system responsible for vision may read images from the blackboard. When it has analyzed them, it writes the observed visual features to the blackboard. This design ensures that the other components are able to gather the information required by accessing it from the global data store without having to communicate directly with each other.

Blackboard architectures were originally developed from the blackboard concept that was used as a theoretical tool in the field of AI research. They were developed into a software architecture and effectively utilized in the HEARSAY-II system (Erman et al., 1980) to implement a speech processing artificial intelligence. Blackboard systems were then enhanced by Hayes-Roth (1985) to provide not just a communal data store but to also provide the control elements required for developing a robotic system. This enhancement formed the foundation of the blackboard architecture used in autonomous robotics.

Blackboard architectures are also frequently used in development projects where limited computation resources are a concern. The RoboCup contest, an annual contest involving teams from research institutions around the world, provides an excellent case study to compare robotic systems that are required to run using limited on-board computational resources (Kitano et al., 1997). Each team competing in RoboCup is required to construct and program robots to perform in a modified soccer contest with the goal of defeating the FIFA world champions by the year 2050. As all of the robots are performing the same task, the differences in programing and construction provide an excellent test bed to compare design choices.

Within RoboCup, there are numerous teams who have independently implemented software architectures. These robotic systems are all designed to achieve the same task. Therefore, the primary differences in the systems are either algorithmic or architectural. Three of the more successful teams using blackboard architectures have released their codebase, allowing a comparison of their implementations. Teams from B-Human (Röfer et al., 2011), UT-Austin (Barrett et al., 2013), and the University of Newcastle's NUbots (Kulk and Welsh, 2012) all have achieved success in the contest and have released code and technical reports on their architectures. Each of these teams use a slightly different implementation of the blackboard system.

Barrett et al. (2013) base their system around a pure blackboard architecture. This system uses a single storage location labeled as "memory" where all information generated by the expert systems is stored. This flat implementation is the original design of a blackboard system with a single, large data store.

Kulk and Welsh (2012) also use a single, centralized blackboard for communication of data between modules. However, several of the sub components have their own private blackboards that

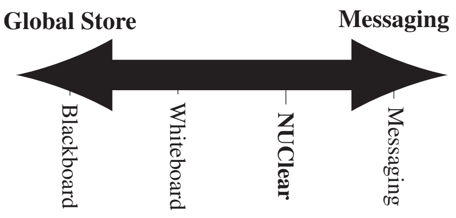

FIGURE 1 | The spectrum of robot software architectures. Most of the architectures used in robotics are some combination of global store and messaging.

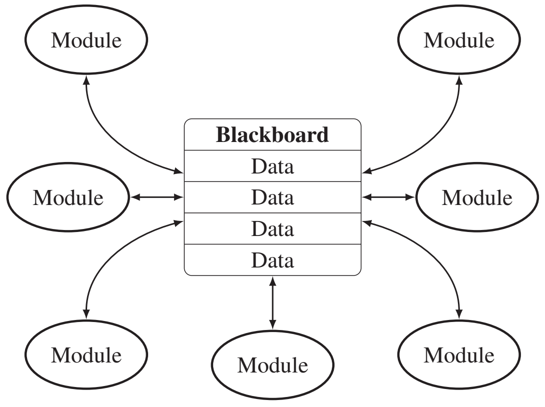

FIGURE 2 | In a blackboard architecture, all modules communicate by reading from and writing to a single global data store. Due to the central data store, all components are coupled through common coupling.

are used for internal communication and memory. This results in a hierarchy of blackboards where each successive blackboard becomes more specialized to an individual task.

The architecture of Röfer et al. (2011) involves numerous individual blackboard elements that are held within separate processes for a particular task. Each of the individual components may have overlapping access to a blackboard. However, each component will only access the blackboards that are relevant to their data requirements. This results in a system where individual blackboards are able to be modified without significantly impacting modules that do not use them.

The B-Human, UT-Austin, and NUbots teams have used blackboard architectures for many years due to the ease of implementation and low computational overhead. A blackboard system has a single location for all data and is easily understood. New states are easily added to a blackboard, and the addition of new data does not require modification to the other systems. This is valuable when performing tests for research purposes as well as being able to add debugging data to the system temporarily.

The system developed by Team KAIST for the 2015 DARPA Robotics challenge (Lim et al., 2015) utilizes a global store system as its primary method of communication. The code executes on multiple processes spread across seven machines in different physical locations. It uses a shared memory system called MPC to communicate data among its processes. The system also uses a message-passing architecture for its vision system based on ZeroMQ. Its control system uses message passing based on POSIX IPC and shared memory. This heterogeneous system has the potential to cause confusion in future development, as it is unclear which architectural style must be used to communicate with a specific component.

In relation to the level of coupling in a system, blackboard architectures perform poorly. In the best case, they exhibit common coupling. All the elements depend on a single data store [Page-Jones (1988), p. 73]. At worst, there can be pathological coupling. This is where modules interact and operate by modifying each others' internal state [Page-Jones (1988), p. 77]. These forms of coupling make it difficult to perform modifications to the codebase due to the flow on effects of modifying the blackboard [Page-Jones (1988), p. 80]. Additionally, despite providing a successful platform for playing soccer, these systems are unable to easily adapt to other roles due to the necessary specialization of the blackboard itself.

Blackboards also present challenges in multithreading situations as individual components are not aware when new and relevant data becomes available. This is of concern in modern multi-core embedded platforms, as modules may miss data updates, read the same data twice, or even read a partially written data element. This makes it difficult to use time-series data effectively, as it requires modifications to both the provider of the data and the blackboard to ensure elements are not missed. Finally, as use cases change and data formats are updated, it becomes necessary to make modifications to all the modules accessing the updated items in the blackboard. This leads to a situation where it is easier to add new data types to the blackboard, rather than refactor the existing data types to improve flexibility in the system.

2.2. Messaging

Message-passing systems are a generalization of a pipeline system where the output from one system is used as the input into the next system (**Figure 3**). Each module creates information and then publishes this information to the rest of the system. Other modules subscribe to these information updates and on receipt of the information, perform their own operations using it.

There have been several successful message-passing architectures developed for robotic systems. Most of these are designed for robots with significant computational resources. However, there are more recent frameworks designed that have considered performance and latency.

OROCOS (Bruyninckx, 2001) is an early open source robotics framework based on a CORBA object broker system. This system allows it to work across multiple languages and processes. It provides a consistent interface for robotics use and is designed to keep code modular for code reuse among systems. However, compared to modern methods, OROCOS is slow (Hammer and Bäuml, 2014) and does not support systems that are not based on object-oriented design patterns.

Dynamic Data eXchange (DDX) (Corke et al., 2004) is a message-passing robotic software architecture developed at the Commonwealth Scientific and Industrial Research Organization (CSIRO). DDX was designed to improve on the slow speed of previous message-based robotic software architectures. These previous systems required messages to pass through a significant amount of the network stack, limiting performance. DDX uses interprocess communication (IPC) to provide a publish/subscribe architecture. As a message-passing system, it suffers from the requirements of serialization of data. Serialization reduces performance significantly as large amounts of data

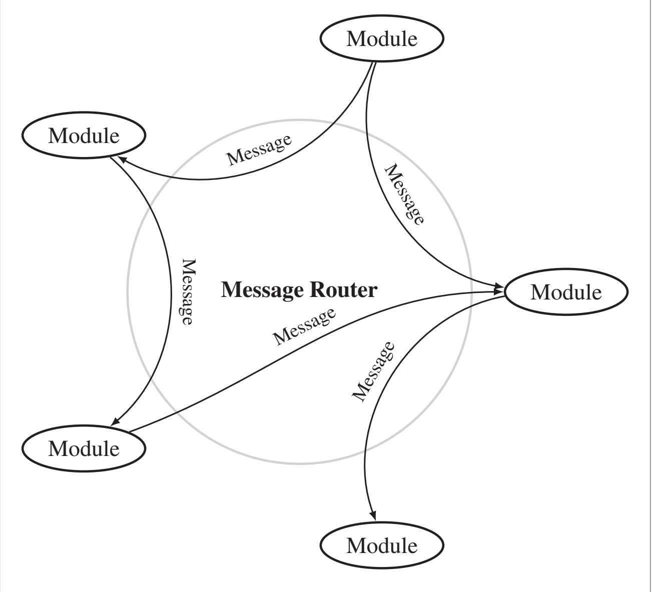

FIGURE 3 | Message-passing systems treat each module as a producer/consumer. Data are sent through a message routing system to subscribers to that type. This message router may be a single entity, or the task may be distributed among individual modules.

must be processed and copied. To resolve this, DDX provides direct shared memory access between modules. However, if used incorrectly, this functionality transforms DDX into a blackboard system and takes on its associated architectural issues. Problems within DDX are also compounded due to the lack of type safety in a system that communicates interprocess. This makes it difficult to ensure that the data received are the anticipated format.

YARP (Metta et al., 2006) is a recent message-passing system for robotics based on the observer pattern. It provides observable entities called ports that can be provided over network and local protocols. YARP utilizes C++ templates to enable customizable serialization of types for transfer. It is able to use shared memory models to reduce the number of data transfers that occur and uses networked methods to distribute information to other systems. YARP modules listen to the observable data either by waiting on a port or by scheduling a callback function to execute when data come in. YARP is an excellent example of a flexible message-passing system.

A common research-oriented robotic software architecture is the Robot Operating System (ROS) (Quigley et al., 2009). It has a centralized node named the ROS Master that is responsible for establishing message routes within the system. Messages do not pass through the ROS Master. Instead, the ROS Master enables nodes to form direct connections. These routes are established using IP networking protocols (TCP or UDP).

Each individual component of the robotic system runs in its own environment without direct knowledge of other components. These components are expert systems responsible for individual tasks, such as localization or visual processing. Communication is handled through predefined messages that are able to be serialized into a transferrable byte representation. Messages that only travel within a single node or systems built to run as a single node, such as ROS nodelets, are able to transfer without copying or serialization.

ROS was designed to provide a platform capable of running code in multiple programing languages on a number of distinct computers to control a robotic system. The flexibility of its programing language and the large number of researchers using this system has made it standard for modern robotics research. This has seen it implemented on numerous robotic platforms ranging from robotic vacuum cleaners to large humanoid robots. The use of ROS has also gained interest in industrial robotics through the ROS-Industrial project (Edwards and Lewis, 2012).

Ach (Dantam et al., 2015) is a messaging system inspired by POSIX message queues. POSIX message queues are suboptimal for robotic systems, as queues create a preference for older data. It caters to multi-processed systems that run on a POSIX operating system and is implemented using a circular array of shared memory that can be accessed by any process that is a part of the system. The shared memory for a particular message is described as a channel. Messages are distributed by adding them to the circular buffer for a channel. Subscribers request new data from the channels they need and then wait for it to become available. Each subscriber has an individual pointer that tracks which messages in the queue have been read. The circular buffer will wrap around and overwrite the old data, potentially causing issues for a subscriber if every data element must be processed. Ach has the advantage as it operates at a low latency due to its use of shared memory. A limitation of Ach is that its use of shared memory restricts its use to a single machine. Additionally, the pull style interface adds complexity for the developer when managing multiple simultaneous requests.

Message-passing systems solve many of the issues encountered with a blackboard system. In relation to the tight coupling of a blackboard, there is no longer any single object that is required to know the entire state of the system. This makes it easier to adapt existing modules for use in a new system. Listening to changes in data allows the system to catch every event as it is modified, removing the need for polling a data store. The ability to distribute the program across multiple threads of execution or even multiple systems makes message-passing excellent for systems with ample computational power. Time-series operations are also possible as every event is accessible without requiring modification to other modules.

Message-passing modules are also able to act as a service. A common pattern is to send a request message and another module will reply with a response message.

Message-passing interfaces will either have a pull interface, where a function must be called to wait for new messages, or a push interface, where a function is provided and executed when data becomes available. In a pull interface, multiple message reads must be multiplexed or have multiple threads listening to receive data. This adds extra complexity to the interface, as there is often increased work required to wait on multiple messages. Push interfaces are simpler to develop, as there is no additional complexity to wait on multiple messages simultaneously. Additionally, push interfaces make multithreading easier as each callback function can be executed on separate thread.

However, there are several disadvantages that exist in these systems that are not present in a blackboard system. A message-passing system must either provide a copy of the data for each subscriber of a message or make all access read only. This results in a performance penalty in the system. Messaging also means that there is no longer a central data store that can be used. Therefore, if a module requires information from more than one message, it must handle the storage of this data itself to access it. This adds significant extra load on the modules, which makes development harder and reduces its performance.

2.3. Whiteboard

Thórisson et al. (2005b) have developed an enhanced version of blackboards in order to utilize some advantages of a message-passing system. Whiteboard architectures have publish/subscribe extensions added to the blackboard in addition to the statically stored data (**Figure 4**). This hybrid system of global storage and message-passing solves several concerns associated with a traditional blackboard system. It is able to use the publish/subscribe features to remove the need for polling in the system, allowing the system to react when data has changed. It also allows the system to perform computations only on new data, rather than repeatedly calling the old data.

The whiteboard design has been used in robotics development. Reykjavik University (Thórisson et al., 2005a) integrates

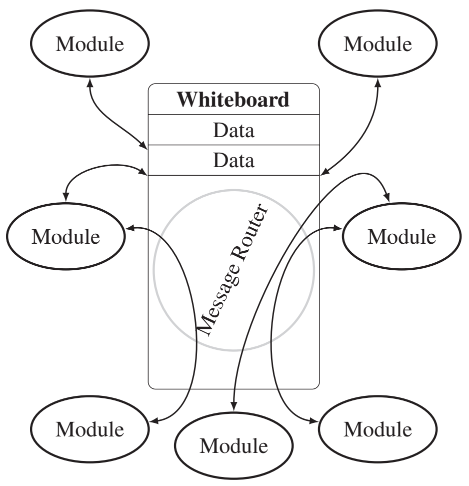

FIGURE 4 | **A whiteboard is a blackboard that is able to have messages passed through it to other modules**. It gains some of the advantages of message passing but maintains the tight coupling of a blackboard.

whiteboard architecture in robotic software, and the Mi-Pal team from Griffith University utilizes a whiteboard architecture to solve issues associated with blackboard architectures (Coleman et al., 2013). Mi-Pal is able to use the publish/subscribe architectural feature to ensure that a module will receive every data update posted. They have recently improved this architecture to provide compile time checking of data types, easing the debugging of the system. Systems that hybridize message passing and global store architectures are a midpoint between the fields of AI that use blackboards extensively and modern software architecture that focus on decoupled design.

Despite these advantages, whiteboard architecture experiences several drawbacks. In a whiteboard system, the weakness of requiring a single blackboard at the center results in common coupling. Even when a stream is used, the stream is stored on the blackboard. This means that modifications to a stream alter the components that use them. The other major issue that arises from using a whiteboard architecture is the use of an implementation that is not a true hybrid of global storage and message passing, but instead two architectures deployed in parallel. In this case, the user either takes on the benefits and weaknesses of a message-passing system or those of a blackboard system.

The performance of the robotic platforms these architectures are deployed on provides insight into why they are utilized. Robots that execute on hardware with tighter performance constraints, such as those with on-board computation, are designed using the blackboard model and occasionally using the whiteboard model. In robotic systems where performance is not restricted, message-passing systems are used to allow greater collaboration between developers. Although message-passing systems provide excellent maintainability, testability, and modifiability, the performance of these systems in relation to latency and overhead is a significant concern when there is limited processing power.

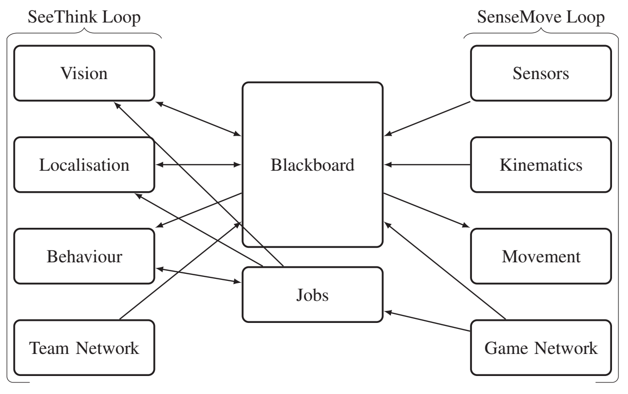

FIGURE 5 | **The information flow from the NUbots' software architecture of 2012**.

2.4. 2012 NUbots' Architecture

The 2012 NUbots' software architecture was designed as an easy to understand and extend object-oriented system (Kulk and Welsh, 2012). However, over time, design-limited module communication and workarounds limited the overall value of the system. This resulted in the system fragmenting and impacted on its quality as the use cases for the system diversified. It resulted in a system that used different assumptions and design patterns. The architecture also caused duplication of effort and the use of similar, redundant classes in various modules due to the lack of a clear architecture. A simplified diagram of the architecture modules and flow of the existing system can be seen in **Figure 5**.

Common with many embodied robot architectures, the 2012 NUbots' architecture revolved around a central blackboard data store. Several modules utilized a global queue known as the jobs system, with most communicating through direct function calls, implicit or global state such as singletons. This diversity in communication resulted in a technical debt and reduced productivity as the system grew. As in order to make modifications to a component, an understanding of the interface, including anything it interacted with or any of the interface's new components, was required. This was problematic as the NUbots' architecture had components that needed to communicate to three or more other components. This required an understanding of a minimum of four different systems before a developer could make any change.

The vision system provides a good example of this unnecessary complexity. The vision system communicated utilizing a two step process. It accessed the sensor system directly and asked it to process a new frame. It then waited on the sensor system to place the frame information on blackboard. Once the frame information was placed on blackboard, the vision system read the information from blackboard and continued its processing. Although the vision system waited for a new frame, the entire robot was blocked and could not make any decisions. It was also important to ensure that no other systems intended to request the latest frame, as doing so would break the robots functionality.

Another unresolved architectural challenge was experienced in the Movement module. The 2012 NUbots' system defined a number of movement handlers, each responsible for a set of movements. This could include kicking a ball or walking. The movement system periodically retrieved all of the jobs in the queue and sent them to the appropriate movement handler. The movement handlers then communicated to the action system to execute the selected motions. This base case did not have any issues. However, it was not possible for any other component to talk to the movement handlers directly *via* the blackboard system. Therefore, at any point in time, any class in the system was a potential candidate for triggering a movement. This made it very easy to create hard to maintain access paths and added to the complexity of the system.

Additionally, each movement handler is indirectly dependent on each other. Movement handlers can lock specific motors and if they attempt to use a motor in use by another system, the action failed. Forgetting to check the ownership of a motor broke the currently executing motion, causing the robot to fall down and possibly injure itself. In order to add a new movement, the code was written for the movement, and the developer was required to understand how all the existing motion modules worked and interacted. This included cases where any of the managers might be triggered, in order to avoid interrupting a critical movement when the motors were locked. Additionally, the developer needed to understand the locking model to prevent accidentally trying to move a motor that should not be moved. These are only two examples of a number of pitfalls within the 2012 NUbots' architecture.

2.5. Comparisons

Previous studies have compared the techniques used in traditional blackboard systems and messaging systems (Orebäck and Christensen, 2003; Magyar et al., 2015; Matamoros et al., 2015). This research concludes that blackboard architectures are easier to maintain and provide greater performance than peer-to-peer systems. However, a significant body of research (Page-Jones, 1988; Pressman, 2005; Beck and Diehl, 2011) has found that loosely coupled messaging systems should provide much greater maintainability and extensibility than globally coupled blackboard systems.

A possible explanation for this discrepancy is that a blackboard system is much easier to comprehend than a message-passing system. This makes it easier for a small blackboard system to be worked on by a developer. However, as the system grows and contains more components, increased complexity should reveal message-passing systems as having the advantage.

3. THE NUClear FRAMEWORK

The NUClear framework was designed to utilize the advantages and address the problems that exist with the architectural styles of message passing and blackboards. The primary advantages of the two competing architectures are the high data availability in a blackboard system and the excellent decoupling properties of a message-passing system. A message-passing system must be used as the primary architectural paradigm to achieve loose coupling. The challenge is to incorporate the advantages that a global store provides into the message-passing system. An ideal system should maintain the loose coupling that message-passing affords, without suffering from data management issues. In order to address these issues and provide a more effective software architecture, a solution called co-messages has been implemented. It more effectively hybridizes the two paradigms in NUClear.

NUClear introduces a modification to how messages are dispatched to the subscribers of the data. In robotic systems, modules will typically have a primary information type that they will perform their operations on. However, they often require additional supplementary data provided from other sources. These additional data must be available when the operations are performed, and they may be created at a different rate to that of the primary data source. Accessing these data at any time is impossible in a pure message-passing system, as the data are only available transiently. If a system requires information created by another module, it must subscribe and then store this information itself to ensure it is available. In NUClear, one of the subscribed types will be designated the primary message type and messages will only be delivered when this message is created by a publisher. The most recent messages of the remaining types are then bound to the subscription, allowing access to the messages without requiring that the modules store the information (**Figure 6**). These additional messages are co-messages. By controlling when messages are received based on the arrival of other messages, the modules in the system are no longer required to manage their own cache of variables.

This method of accessing data is distinct when compared to multiplexing multiple message channels into a single input. In comparison to using select or poll on multiple channels, co-messages do not introduce a cost for messages that are not used. This is often the case when the rate of the messages do not match. When one message occurs more frequently than another, a system using select or poll system must inspect and discard each of these updates when they are not relevant. Instead, in NUClear, those messages are fetched on demand when the primary data

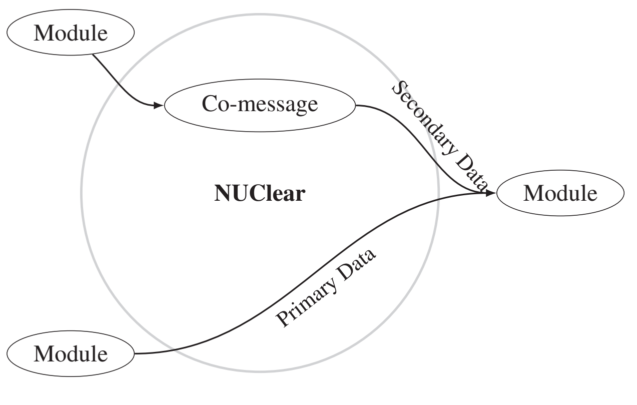

FIGURE 6 | The NUClear co-messaging system. The latest version of each message is stored in NUClear. A module requests a primary data type, along with the most recent version of a secondary data type. When these primary data are generated, the stored most recent copy of this co-message is bound into the callback. This affords the module a more expressive interface for gathering messages.

type is received, reducing the overhead both computationally and for the developer.

Additionally, as the latest version of each message must be stored in order to be bound when needed, a virtual global store is created from these messages. These globally stored data are not an explicit component, but a byproduct of the architecture. It does not increase the coupling between modules. Therefore, the developer is not required to write code for data elements in this global store. This resulting architecture has the advantages of loose coupling from the message-based system, while gaining the advantages of high data availability provided by a centralized store.

The virtual data store is also able to be explicitly accessed, providing direct access that is similar to a blackboard. As the messages are persistent, they are able to be accessed without setting up a listener for the message. This form of access is discouraged, as it increases the coupling in the system. However, this provides an advantage to NUClear in its compatibility with code written for other architectures. NUClear is able to accommodate the access patterns of any of the three other architectures with only minimal modification to the components themselves.

A key requirement of a multi-platform robotic system is the replacement of components with functional equivalents. For example, it should be possible to replace the system that reads camera frames from the hardware with one that reads frames from a file without changing the rest of the system. However, another important aspect of robotics is the requirement for real-time interaction with the world. This requires optimized code paths. As identified in the background research, the architectures excel in one of these categories, but often trade speed for decoupling.

The majority of mature robotic systems are developed with a message-driven architecture. ROS has a serialization penalty associated with message passing, as communication between modules is achieved using sockets. This is unacceptable for low-power embedded platforms, such as the Darwin-OP. Other robot systems, such as Dynamic Data eXchange and Pack Service Robotic Architecture, also experience similar issues. These contemporary robotic systems have traded performance for message passing and cross-language compatibility, anticipating that future hardware performance will accommodate them.

3.1. Simple API

NUClear is designed to have a simple and intuitive interface that requires minimal training to use. It is designed for use by second year software engineering students who have a basic understanding of C++. There must be a well-defined function for the two key features of the architecture: sending and receiving messages. Sending is accomplished using the *emit* function, whereas receiving is accomplished using the *on* function. NUClear provides a small domain specific language (DSL) for easy access to common message pattern use cases. This DSL is designed to match an English description of the task if possible. Having a small API improves understandability and reduces the learning curve for using the NUClear system. This makes it easier for new programmers to learn and use NUClear.

In addition to the DSL words that are already in the system, NUClear is designed to be able to extend its vocabulary with new words developed by the user of the framework. This allows common functionality to be provided across modules specific to the needs of a system. Internally, all NUClear DSL words are implemented using the same extension system.

3.1.1. Domain-Specific Language

The following is a list of the DSL words that are included and most commonly used in NUClear along with a description and an example of their use.

on	`on<...>(runtime...).then(function);`	The on DSL word is the wrapper for every subscription in NUClear. NUClear uses this to wrap the template descriptions of the subscription's purpose. The other DSL words are entered as template arguments to this function, with any runtime arguments passed as function arguments.
emit	`emit(message)`	Emit is the function that handles the publish part of the messaging system. It takes data and forwards it to the functions that have subscribed directly (routed at compile time).
Trigger	`on<Trigger<Type>>()`	Trigger statements set up the callbacks and execute when the type is emitted. It flags the used data type as a triggering (primary) data type. When this callback is executed, it will pass the data that were emitted.
With	`on<Trigger<TypeA>, With<TypeB>>()`	With statements describe additional information that is used by the callback. The provided function will not be executed when these data are emitted. However, when this function is executed, the latest copy of this data will be provided.
Every	`on<Every<10, milliseconds>>()`	Every statement fills the role of periodic callbacks in the system. When an every statement is used, the function will execute at that rate.
Always	`on<Always>().then(function);`	Always is used in the rare case that functions must continually execute as fast as possible. It allows the system to terminate as a whole when it shuts down by ending the execution of this function.
Single	`on<Trigger<Type>, Single>()`	Single is a DSL keyword that ensures that only a single instance of a message will be executed at one time. When additional messages of the same type are given to this function, they will be dropped.
Buffer	`on<Trigger<Type>, Buffer<3>>()`	Buffer is the general case of the Single keyword. It ensures that only the requested number of messages will execute simultaneously. When additional messages beyond the requested number are given to this function, they will be dropped.
Sync	`on<Trigger<Type>, Sync<Group>>()`	Sync is used to ensure mutual exclusion between several functions at a scheduling level. Rather than blocking a thread on a mutex, it will delay execution until it has exclusive access among a group of functions. Additional messages of the same type are queued for execution unless combined with single.
Priority	`on<Trigger<Type>, Priority::HIGH>()`	Priority can control the response of the callback. It will control both the priority used to determine the scheduling order in the thread pool and also the priority of the thread it will execute on.
Startup	`on<Startup>()`	Functions with this word will execute at startup.
Shutdown	`on<Shutdown>()`	Functions with this word will execute at shutdown.
Configuration	`on<Configuration>("File.yaml")`	This allows a program to watch a file in a configuration directory and be provided with the latest version of the file when it changes. It is used to keep configuration up to date.

Optional `on<Trigger<TypeA>, Optional<With<TypeB>>>()`
Optional allows statements to signify certain requested types as optional. In a traditional co-message call, if both messages are not available, the function will not run. If optional is used, these functions can run with an empty second argument.

Last `on<Last<10, Trigger<Type>>>()`
Last instructs NUClear to cache the last messages that were emitted. When the function is called, it will receive all of the collected messages.

IO `on<IO>(file_descriptor)`
Used to interact with file descriptors and execute when they are read/writeable. This is used for communicating with serial devices as well as network ports.

Network `on<Network<Type>>()`
NUClear provides a networking protocol to send messages to other devices on the network. This can be used to make a multi-processed NUClear instance, or communicate with other programs running NUClear. The serialization and deserialization is handled by NUClear.

TCP `on<TCP>(port)`
TCP allows a program to make a callback on TCP activity, listening on a port.

UDP `on<UDP>(port)`
UDP allows a program to make a callback on UDP activity, listening on a port. It also supports listening to UDP broadcast and UDP multicast sockets.

3.2. Low Performance Penalty

In the context of embedded or low performance hardware, it is essential that message routing is performed as quickly as possible. Other message-passing systems use technologies that incur performance penalties from message serialization and copying. Instead, NUClear uses shared memory for messages passed within a single process. This removes the cost of serializing a message and can greatly improve the performance of large messages.

It is also important to minimize the time it takes to dispatch a message. To achieve this, NUClear uses template meta-programing to establish message routes at compile time. Generally, when a message is sent in a message-passing system, there is a message broker that executes code to find subscribers who are interested in the message. Alternatively, in simpler systems, there is a message bus that all subscribers listen to and gather messages they are interested in. NUClear eliminates this cost by evaluating the route messages take at compile time. The result is that when messages are dispatched from a module they are directly sent to the modules that require them. This reduces the cost of routing the message to acquiring the required data and directly calling the subscribers callback function. The downside to this technique is that it is only applicable when running within a single process.

NUClear also has an additional latency improvement for modules arranged in a pipeline structure. This improvement is used when a module emits a message at the end of its callback, and that message is only consumed by only one other module. In this case, NUClear is able to continue and execute the following code directly rather than returning to the thread pool. This greatly reduces the latency between the two modules, as when combined with compile time routing, there is little that occurs between the executions.

Additionally, NUClear is able to perform compile time message memory allocation using template meta-programing. It uses the information to preallocate the space before the program runs. This allows it to scale to any number of messages while maintaining $\mathcal{O}(1)$ dispatch look-up time. The mechanism used to allocate messages also enables optimizing compilers to use the knowledge of message memory allocation to apply a number of powerful optimizations to how messages are sent.

3.3. Simple Utilization of System Resources

Another requirement derived from resource-constrained environments is the need to easily utilize the full power of the hardware. This is primarily achieved by introducing transparent multithreading that automatically uses every CPU core available on the system. Transparent multithreading in NUClear utilizes a thread pool that has enough threads to saturate every CPU core. Using this thread pool, NUClear is able to schedule each execution of a callback function to a different thread. This allows the system to utilize all of the cores without the developer needing to interact with threads directly.

When a message is sent in the system, the central coordination object, known as the PowerPlant, takes ownership of it. From this point forward, no modules can modify the message. It is provided to other modules with read-only access. The PowerPlant then executes callbacks that, known as reactions, are subscribed to this message. Each reaction receives an immutable reference to the original message and can perform any read operations on it.

The immutability of the data makes transparent multithreading in the system easier. If multiple reactions want to read the data, and if it can be guaranteed that they do not modify it, then the reactions can be run in parallel without concern for race conditions. By forcing immutability, all threading logic can be moved directly into NUClear, allowing developers to concentrate on their modules instead of threading problems. This technique of using immutable data to allow easy multithreading systems has been proven in programming languages, such as Erlang and Elixir.

In most cases, multithreading will be completely transparent. However, it is still important that developers understand that they are working in a multi-threaded environment. If a single module shares data between two reactions and those reactions run in parallel, then the shared data will need to be secured with a thread-safe mechanism such as a mutex. NUClear also provides functionality in its DSL to let the user specify that certain reactions should not be run in parallel. Specifically, it provides the word *Single* to ensure that only a single instance of a reaction is running and to drop any future messages. It also provides the word *Sync* to ensure that only one of a group of reactions is running and to queue the remainder. These can be used to solve threading concerns.

Differences exist between a multi-threaded system and a multi-processed system. In robotic systems, it is common to run in a multi-processed mode rather than multi-threaded. This allows the system to be distributed across multiple physical hardwares, making more efficient use of available resources. It can provide a level of crash safety. If a single process in the cluster crashes, it does not result in an entire system crash provided the remaining modules can run independently. Multi-threaded systems also have this property if the threads the code executes on are managed correctly. This requires additional work by designer of the multi-threaded system.

Multi-processed systems also have disadvantages. Multi-processed systems do not always run in a shared memory environment, as they may be distributed across machines. They must serialize and copy data to the destination nodes. This adds an increase in latency between modules and when there is a large amount of data to be transferred, this can have significant implications.

Multi-processed systems also suffer from an increase in the context switching time. When the operating system in a multi-processed system switches from one process to another, it changes its virtual memory space. This requires the translation lookaside buffer to be dumped. Multi-threaded processes share their virtual memory space, allowing the buffer to be retained.

NUClear is able to run in a multi-threaded mode, a multi-processed mode, or a hybrid of the two by grouping modules into individual processes. However, when it runs in a multi-processed mode, it loses many of its advantages from compile time message routing. This is a necessary side effect, as the compiler can no longer optimize code paths and it must pass messages by serializing them over a socket.

3.4. Time-Series Data

A common need when interfacing with real-world electrical systems is to keep a record of recent data for validation and decision-making. The simplest example of this is de-bouncing the electrical noise in a button press. To perform this task, the on/off state is monitored over a time-series, with an internal stateful check deciding whether the switch is deemed to have closed or not. These types of tasks commonly access the last *n* instances of created data when performing an evaluation.

NUClear handles time-series data at an architectural level by increasing the number of previous messages that are stored latent in the system. This functionality is provided through the Last keyword.

A blackboard system is unable to receive a history of elements, as there is no notification of when data are updated. This may result in duplicate and missed data depending on the rate of polling. Rather, the functionality must be added by the producer of the data. They must store the additional data on the blackboard unaware of when it is not needed.

Message-passing systems are able to provide time-series data, as the arrival of messages allows the subscriber to maintain a history of the last few messages. This requires the subscriber of the messages to maintain their own data store of the most recent messages.

In a whiteboard system, it is possible to obtain time-series data using the same mechanism as message passing, or blackboard, depending on the stored data. However, they are also able to use the publish/subscribe channel in order to remain informed of changes to static data, allowing the data to be copied.

3.5. Soft Real-Time

Using the functionality provided by Every and Priority, NUClear can operate as a soft real-time system. One reason NUClear is successful at operating at soft real-time is its compile time dispatch of messages. When the binaries are compiled, the periodic functions are compiled with them. This allows the periodic functions to operate with the jitter and accuracy that the operating system providing the timing is capable of. The jitter in NUClear's periodic Every function was measured at 80 μs when triggering at a rate of 1 kHz.

Additionally, as NUClear routes messages at compile time, it can achieve lower end-to-end latency between modules. This is a reduction in time between dispatching and receiving a message. This lower latency can assist systems that have strict timing requirements such as hardware feedback loops.

A simple test system was constructed in NUClear and ROS that timed and transferred an empty message from end to end. This should eliminate any performance differences due to serialization and copying of information. The tests were completed with and without the CPU being loaded to 100%. All tests were completed on an Intel i7 1.8 GHz with 16-GB RAM running Ubuntu 14.04 Desktop (Linux kernel version 3.13.0.74.80). This test generated 100,000 data points for each of the sets. The results are shown in **Figure 7**. A TCPROS node was included in the test, as this is the default communication method within ROS. To use ROS inter-node communication requires special setup using ROS nodelets, which may not always be possible. The inclusion of TCPROS also provides a reference point for network communication speeds.

These results show that NUClear is faster at routing messages than ROS. In fact, the latency between modules in NUClear is faster than routing within a node in ROS. Interestingly, NUClear's performance improved when using a thread pool under system load. This is believed to have been caused by the delay in waking up a sleeping thread. When the system is already under load, these threads do not go to sleep, which reduces latency. More rigorous testing of the compile time message routing system is planned for future research.

3.6. Statistics, Logging, and Traceability

In complex systems, it can be difficult to determine a system's operational state. NUClear is designed to support powerful statistics and logging tools. NUClear stores runtime statistics about each module and provides mechanisms to receive the information

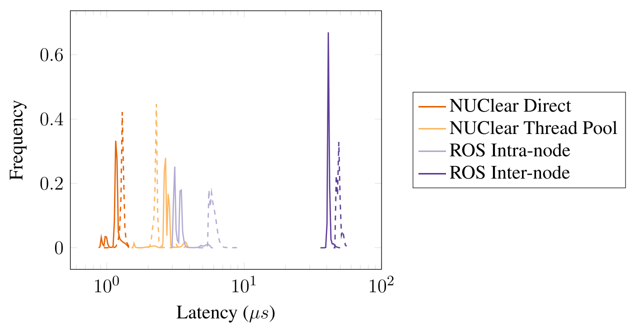

FIGURE 7 | Frequency distribution of end-to-end latency for NUClear *via* a thread pool, NUClear directly sending messages, ROS within a single node, and ROS between nodes on a single machine (TCPROS). Dashed lines represent the test done with the system running at 100% CPU load.

logs on a per-module or per-event basis. These features provide useful information that assist with debugging and understanding the robot's system.

If an error occurs, it is possible to capture the input that caused the error and replay it on the module. This is possible as the architecture itself is aware of the information used by each callback. Each callback requests the list of all messages required to complete its task. This represents the current state of the system. Since these messages can be captured at the point of callback, data that cause errors can be captured and examined more closely. This recording functionality can be used to develop a powerful array of tests that accurately reproduce real-world scenarios. These features are compiled on demand, with unused features not impacting on the performance of the system.

A networked visual debugger, NUsight, is also built into the NUbots' codebase. This system supports streaming operational data in real time to a web-based visualizer. The visualizer can perform real-time charting of time-series and 3D visualization of the robot's believed state. These features are easy to use in modules in the system due to NUClear's extensibility. Additional DSL words are added making code to view internal state only to exist when needed and easy to use.

4. NUClear EVALUATION

The NUbots' codebase is used as an example to quantitatively assess the improvements provided by the NUClear framework. This codebase was chosen as it is a large (~80,000 LOC) codebase that previously implemented a blackboard architecture that can be used for comparison. Additionally, this codebase has multiple distinct binaries with each performing a distinct functional or testing role. Each of these roles includes a different set of modules. Due to the compile time message passing of NUClear and the fact that it uses co-messages, it is possible to extract the graph of message relationships between modules from each compiled binary. Once extracted, it is possible to transform this graph into the equivalent graph for a more traditional message-passing system, such as ROS or YARP, by taking all non-triggering data and creating a separate subscription event for it. This adds a new subscription for this type, as well as a local cache to store the message for when it is used. Only one cache is needed per module, so it is only counted for the first instance of a type.

4.1. Interface Size

Using co-messages in NUClear reduces listener code compared to other message and event-based systems. Rather than having a separate subscriber and cache for each data type, there can be a single subscriber for multiple data types without needing a separate cache. This directly reduces the number of functions that must be written to handle these cases. The difference in the number of functions that must be written can be seen in **Figure 8**. In this graph, the number of subscription handlers is shown for a NUClear system, compared to the theoretical equivalent message-passing system for each module.

Figure 8 shows that while some modules have the same number of callbacks in both systems, there are many cases where up to double the number of subscription handlers must be written for a traditional message-passing system. A large number of these handlers will also be caching the variable for use by the primary function. When refactoring these, caching handlers can be forgotten and contribute to dead or poorly documented code.

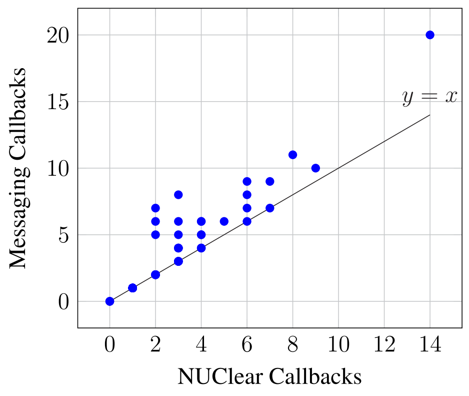

FIGURE 8 | The number of functions needed by a module in NUClear compared to a messaging system. Each point is a module in the NUbots' codebase. The height above the $y = x$ line indicates how many additional callback functions must be written in a message passing system.

Direct comparison with a blackboard system is difficult, as it does not use callbacks and the data must instead be polled. However, variables can be read at any time in a blackboard system. This results in various problems, including thread safety and synchronization.

Whiteboard-based systems allow a similar level of performance to the NUClear system in relation to the code efficiency of each module, as data are accessible at any time. However, whiteboards move the burden of dead and poorly documented data from the individual modules to a large, central data store. Over time, this can become difficult to maintain.

In comparison with these systems, the callbacks in the NUClear system specify which data are needed for a specific operation. This removes the responsibility of implementing cached data storage from the system and results in a system with significantly less implementation overhead, while retaining the speed benefits of universally accessible data structures, such as blackboard architectures. It also maintains most of the modularity and flexibility of traditional message-passing systems, such as ROS.

4.2. Memory Usage

NUClear provides a reduced memory footprint in comparison to message-passing or blackboard-based systems. This is because it is able to determine and store only the data that are live at any time. **Figure 9** shows the number of additional cache variables required for a messaging implementation of each of the binaries present in the NUbots' codebase. These additional variables are required as each module must cache any data type that arises as a result of a non-triggering message ("With" messages in the NUClear framework). As a result of NUClear managing all of these variables centrally, additional caching variables are not required in a NUClear system. This can be useful for memory constrained systems.

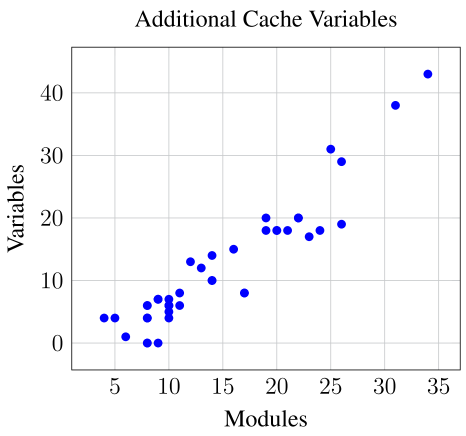

FIGURE 9 | The number of additional cache variables that would be required if the NUbots' codebase was implemented in a message-passing system. These additional variables would be required to store data that were requested as a non-triggering type. Each point represents a single binary from the NUbots system.

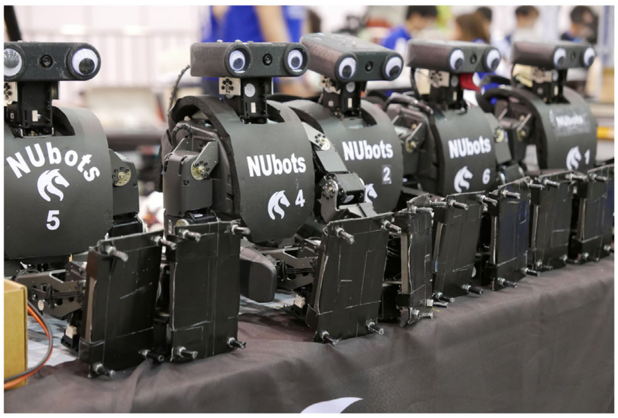

FIGURE 11 | Modified Darwin-OPs of the 2015 NUbots team equipped with new 3D-printed heads using a higher resolution camera, padded jackets to soften falls, and soccer studs to improve grip on artificial grass.

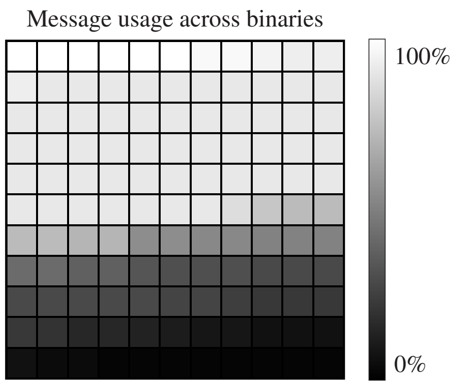

FIGURE 10 | The distribution of usage for the 121 message types across binaries in the NUbots' system. This serves as an indication as to how much memory an equivalent blackboard system would use.

An indication of memory usage in a theoretical blackboard system can be seen in **Figure 10**. In the NUbots' system, there are currently 121 message types displayed as an 11 × 11 grid. However, only a subset of these messages is used in any binary. The binary that uses the largest number of messages only uses 110. If a blackboard were to be used, all of these messages would be stored for every binary. This can also be seen by comparing the system to the previous blackboard based NUbots' system. By comparison, NUClear is able to determine which messages are needed for listeners at compile time and does not store unused data types.

4.2.1. Cache Computational Overhead

In a message-passing system, there is an additional cost incurred. When a message is received in a traditional messaging system and is used as additional data, it must be cached. Writing this cache costs performance from copying the data to a local variable for storage. Large messages can have a significant cost if they must be repeatedly copied to local variables. Additionally, when the data rates are not matched, much of the copied data are never used. For example, in the NUbots' codebase, the sensors are read at a rate of 120 Hz and images at 30 Hz. Ninety sensor messages per second would not be used, but in a message-passing system would still be cached. When the code is running on a system with limited CPU power, these costs reduce the available computational resources for other tasks.

5. TARGET PLATFORMS

The current primary target for the NUClear architecture is the Darwin-OP platform (Ha et al., 2011). Modified versions of the Darwin-OP that have slightly different kinematics and camera sensors are also supported, pictured in **Figure 11**. These platforms have twenty degrees of freedom provided by ROBOTIS serial controlled servomotors, a six degree-of-freedom IMU and a webcam for sensing. On-board processing is provided by an embedded Atom z530 processor running at 1.6 GHz.

The NUClear architecture has also been used in the RobotX Maritime challenge on an autonomous marine platform. This platform involved the use of an embedded control system and a set of four global shutter cameras with wide field of view lenses. Due to the system modularity that NUClear allows, large parts of the vision system were taken from the NUbots' soccer codebase with minimal changes.

Future plans include deploying NUClear to the NimbRo-OP platform (Allgeuer et al., 2015) and autonomous quadcopters. These will be interesting platforms to test the flexibility of the NUClear architecture. The larger Nimbro-OP platform also provides opportunities for direct comparisons with a ROS-based framework. This future work will support a comparison of maintainability and efficiency across a spectrum of robot software architectures with the factor of hardware variation removed.

6. CASE STUDY – NUbots' CODEBASE

The architecture implemented in NUClear allows a humanoid robot to perform a variety of tasks within structured and semi-structured environments. Taking advantage of the message passing and storing capabilities of the system and building on these in each of the subsystems has allowed a flexible, but principled approach to robot control. A simplified flow of information through the NUbots' system is given in **Figure 12**.

The core of the NUbots' system is a see-think-do type information flow. This is common to previous architectures. It allows for information to be taken in from the surroundings and for planning and movement to take place. In addition to innovations and improvements within the subsystems of this loop, the architecture also allows a generic fast-path that enables all motor skills to react to changes in the environment quickly. This can provide balance and reflexive protection skills more easily than in previous systems, with reaction speeds that are faster than current message-passing systems.

6.1. Vision Pipeline

Many challenges for mobile and embedded robotics involve the use of computer vision to sense and navigate environments. In recent years, environments for humanoid robotics challenges have transitioned from a well-defined color and pattern-based structure toward more realistic environments. As these challenges increasingly reflect the real-world, semi-structured environments where shape and context give vital additional information about the world must be considered. The machine vision community has developed many approaches to structure-based detection. However, most of these cannot yet be computed in real-time on lower power embedded systems. The vision pipeline of the NUClear system represents an efficient compromise between structure-based and color-based systems by using color classification to find regions of interest, followed by edge-based shape fitting to find particular objects (Quinlan et al., 2004; Henderson et al., 2008; Houliston et al., 2015).

The vision system used in the NUbots' system shows one significant difference to embodied vision systems, popular in the ROS message-passing system. The source code provided by Allgeuer et al. (2015) uses a monolithic vision module in ROS, as the performance penalty for message-passing (particularly for large data types such as images) for ROS communications is considered quite high. This can make message-passing vision systems in ROS slow to process and report changes in the environment. Due to the much faster message dispatch, the NUClear architecture is able to provide both better modularity and performance by parallelizing module execution where possible. Additionally, the multithreading controls provided by the NUClear architecture remove the need for detecting when the system is overloaded and lagging. This removes the need for implementing queues and high-watermark throttling.

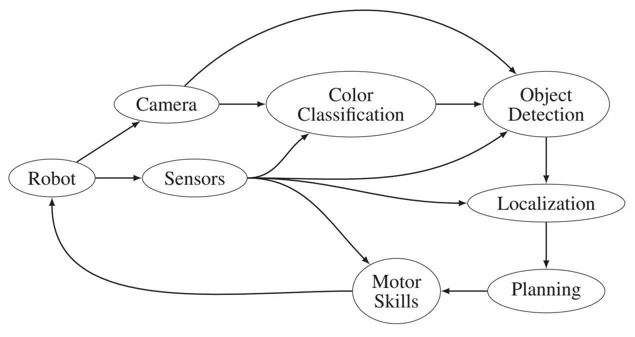

FIGURE 12 | High level overview of the NUbots' information flow loop. Each of the elements represents a collection of modules that work together to achieve the goal of that element.

The first stage in the vision pipeline involves reducing an image from raw pixel data to color and edge information. Using the thread-aware features of the NUClear architecture, the vision pipeline ignores incoming images when the compute load is too high. It is also possible to define multiple input camera streams, each of which processes and drops frames independently and fairly. This allows smooth operation for multi-camera robots, as well as avoids loading down the system with heavy image processing. The image color classification itself is performed using a look-up table that converts pixel values into symbolic colors.

Several investigations have been made into automating and improving color classification within these systems (Henderson et al., 2008; Röfer et al., 2011). The current system furthers this work by dynamically adapting to light and color conditions based on detected objects and regions. Using the NUClear framework's Priority keyword, it is possible to run the dynamic color adaptation as a low priority process that does not interfere with the system's normal running or process outdated data.

Objects are detected using a collection of independent color and shape-based detector modules (Murch and Chalup, 2004). As the inputs and outputs for detector modules are defined as messages and the NUClear system is compiled as a collection of modules, it is very simple to include or swap out detectors for various testing purposes. This strength is similar to the utility that ROS provides; however, message dispatch times are significantly reduced. This allows real-time tracking of relatively fast-moving objects such as rolling balls. NUClear also provides a functionality to disable and enable the execution of modules at run time, allowing detectors that are not needed for the current task to be switched off.

Versions of image color classification have been used within the context of robot competition environments for some time. However, improvements in camera sensors and noise filtering have reached a level of robustness suitable for deployment in more difficult and dynamic environments. As such, this system was also used in the RobotX Maritime Challenge with minimal modification.

6.2. Localization and Mapping

With the modularity of the rest of the NUClear architecture, it is important when developing localization components to allow for a range of inputs and drop-in replacements. Localization modules take visual detector observations and orientation sensor updates as inputs. The structure of observations provides an angle and distance from the robot in three dimensions. This supports inputs to be as diverse as users' faces to marker symbols on the ground, while using the same interface. Multiple estimations with differing confidences are allowed for each detection, as there may be more than one hint in the image about the distance of the object. The significant difference between NUClear and other message-passing architectures for localization and mapping is that NUClear removes the need for maintaining caches and matching sensor timestamps when performing data fusion updates reducing the amount of code required. This improves the clarity of the code and algorithms used, as well as improving efficiency slightly.

6.3. Planning and Actuation

The NUbots' system utilizes an extended form of subsumption logic (Brooks, 1986) to arbitrate access between skill modules and the robot's hardware. The system operates through the registration of each module at system start-up (which can be specified by the NUClear Startup event), along with their current subsumption priority and the hardware components they wish to control. The subsumption controller then allocates resources to the registered modules based on which module has the highest priority for a subset of the robot's limbs. The subsumption priority of a module can be changed in real time by request from that module, allowing the system to dynamically allocate control as priorities change.

The use of a flexible, thread-aware message-passing system with very fast dispatch times supports the implementation of modular reflexive behaviors and failsafes without impacting on the implementation of the rest of the system. This has been used in the NUbots' system to implement universal reflexive responses. An obvious candidate for reflexive behavior on a bipedal platform is the implementation of a protective reflex to reduce damage from falls. This reflex acts on the filtered gyroscope and accelerometer data produced from the sensors without going through any other processing or delays. If the robot is falling, the reflex takes highest priority on all limbs until the accelerometer indicates that the robot has come to rest.

The fall reflex was originally designed to mimic humans, putting arms out to soften the impact. It was found that throwing the arms forward with elbows bent and then deactivating the motors to become fully compliant was effective at reducing the impact to the robot's body. These types of reflexes are only effective if action is taken almost at the instant the robot realizes it will fall; otherwise, there will not be enough time to effectively reposition the body. For this reason, slower message-passing systems may not be able to effectively implement reflexes as independent modules.

Head behavior is one of the unique challenges involved with humanoid robotics. If a robot is to be truly humanoid, it should not use a visual system that has a field of view greater than that of a human. This limits the amount of the environment that can be perceived at any one time. Humans meet this challenge with two key behavioral systems: head movement and eye movement. The movement of the head adjusts the coarse field of view, while the much faster movement of the eye defines where high detail is perceived (Duchowski, 2007).

The NUbots' head behavior system has been developed to mimic the blur-reduction of human eye behavior, in particular, the vestibulo-ocular reflex (Fetter, 2007). The vestibulo-ocular reflex is the mechanism by which stationary objects in the world are stabilized on the retina of the eye during head movement. This works through a tight feedback loop with both vestibular information from the balance system and visual information. Humanoid robots have a similar problem with image stabilization. When the robot's body rotates, typically the head rotates with it. This can blur the camera images. The NUbots' system takes the most recent orientation information, in the form of a rotation matrix, and uses this to set a constant look direction in the global reference frame regardless of the robot's motion. As with the fall-protection reflex, this type of reflexive behavior requires very fast end-to-end response times to be effective.

6.4. Robot Learning

Learning is a fundamental human skill. As online planning and machine learning algorithms advance, it is important to incorporate features that simplify their implementation on any robot software architecture. Distinctions must be made between online learning, offline learning, on-board learning, and learning processed on an more powerful external system. Most message-passing systems support network communication and data output for offline learning. The NUClear architecture also provides a high efficiency framework for implementing embodied and online learning systems, as well as methods for sharing, storing, and visualizing the data produced by these systems.

Of particular use in the context of learning is the NUClear Last keyword. This provides a cached stream of events to be processed during learning updates. This simplifies the implementation and maintenance of embodied movement-based optimization and learning algorithms, which often operate in batch mode on a stream of evaluation data at the end of a movement (Kalakrishnan et al., 2011; Budden et al., 2013). As such, NUClear simplifies the development process for many embodied machine learning algorithms when compared to previous architectures such as the platform of Kulk and Welsh (2012). Due to lower system overhead and faster inter-module communications, NUClear is also more efficient than traditional message-passing architectures such as ROS when fulfilling this role. NUClear has key advantages over other systems due to its co-messaging and built-in keywords that allow simpler implementation of machine learning algorithms.

7. CONCLUSION

Message-passing systems are an ideal solution to robotic architectures. This is supported by the popularity of the ROS research architecture and other message-passing architectures. Systems based on message passing have been at the forefront of software architectures for high functioning robots for the last decade. The isolated components operate independently and when coupled with a message-passing system, they can be altered and replaced with much greater ease.

The limitations that prevented these architectures from being deployed on robots without sufficient performance have been removed through the development of the NUClear framework. The NUClear framework is able to operate at speeds that approach native function calls, while also providing transparent multi-threading and type safe data storage. The ability to access multiple message types simultaneously greatly speeds the development of new modules, as this access pattern mirrors humanoid robots' sensor systems. In contrast to other message-passing systems, it allows programmers to specify the global data requirements as a part of the listener declaration, rather than constructing a separate listener for each piece of data required. These data are guaranteed by the NUClear framework to be accurate and act as a clear dependency definition in one place, which improves the knowledge of the system.

These advantages make the NUClear system a valuable proposition for research robotics. The advantages also suggest that further investigation should be taken into message based systems that contain a global message store. These systems have properties that greatly simplify the fusion and processing of data streams that are required for modern humanoid robotics, which could lead to more maintainable and flexible systems in the future.

AUTHOR CONTRIBUTIONS

The system has been developed over several years where the students of the team (TH, JW, JF, and MM) contributed code development with the software architecture work performed by TH. The academics in the team (SC, AM, and YL) contributed management and supervision. The lead authors of the current manuscript are TH with assistance from JW. Information for sections of the case study was provided by JF and MM.

ACKNOWLEDGMENTS

The authors are grateful to all previous members of the NUbots team who contributed to development of the software base since 2002. Additional key developers and contributors at various stages of this process were Brendan Annable, Monica Olejniczak, Peter Turner, Aaron Wong, and Jake Woods. The authors thank Alex Biddulph for his assistance running the timing comparisons with ROS. The authors are also grateful to Ellie-Mae Simpson for assistance proofreading this manuscript.

REFERENCES

Allgeuer, P., Farazi, H., Schreiber, M., and Behnke, S. (2015). "Child-sized 3d printed igus humanoid open platform," in *IEEE-RAS International Conference on Humanoid Robots (Humanoids)* (Seoul: IEEE).

Barrett, S., Genter, K., He, Y., Hester, T., Khandelwal, P., Menashe, J., et al. (2013). "UT austin villa 2012: standard platform league world champions," in *RoboCup 2012: Robot Soccer World Cup XVI, Volume 7500 of Lecture Notes in Artificial Intelligence (LNAI)*, eds X. Chen, P. Stone, L. E. Sucar, and T. V. D. Zant (Berlin; Heidelberg: Springer), 36–47.

Beck, F., and Diehl, S. (2011). "On the congruence of modularity and code coupling," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11* (New York: ACM), 354–364.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* 2, 14–23. doi:10.1109/JRA.1986.1087032

Bruyninckx, H. (2001). "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation, 2001. Proceedings 2001 ICRA*, Vol. 3. (Piscataway, NJ: IEEE), 2523–2528.

Budden, D., Walker, J., Flannery, M., and Mendes, A. (2013). "Probabilistic gradient ascent with applications to bipedal robot locomotion," in *Australasian Conference on Robotics and Automation 2013 (ACRA 2013)*, eds J. Katupitiya, J. Guivant, and R. Eaton (Australian Robotics & Automation Association).

Coleman, R., Estivill-Castro, V., Fernandez, E., Geffner, H., Gilmore, E., Ferrer, J., et al. (2013). *Mi-Pal Team Description 2013*. Available at: http://www.informatik.uni-bremen.de/spl/pub/Website/Teams2013/MiPAL.pdf

Corke, P., Sikka, P., Roberts, J. M., and Duff, E. (2004). "DDX: a distributed software architecture for robotic systems," in *Proceedings of the 2004 Australasian Conference on Robotics and Automation (ACRA 2004)*, eds N. Barnes and D. Austin (Australian Robotics & Automation Association).

Dantam, N., Lofaro, D., Hereid, A., Oh, P., Ames, A., and Stilman, M. (2015). The Ach library: a new framework for real-time communication. *IEEE Robot. Autom. Mag.* 22, 76–85. doi:10.1109/MRA.2014.2356937

Duchowski, A. (2007). *Eye Tracking Methodology: Theory and Practice, Second Edition*. Springer-Verlag London Limited.

Edwards, S., and Lewis, C. (2012). "Ros-industrial: applying the robot operating system (ROS) to industrial applications," in *IEEE Int. Conference on Robotics and Automation, ECHORD Workshop*. St. Paul.

Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. (1980). The hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty. *ACM Comput. Surv.* 12, 213–253. doi:10.1145/356810.356816

Fetter, M. (2007). Vestibulo-ocular reflex. *Dev. Ophthalmol.* 40, 35–51. doi:10.1159/000100348

Ha, I., Tamura, Y., Asama, H., Han, J., and Hong, D. (2011). "Development of open humanoid platform DARwIn-OP," in *Proceedings of SICE Annual Conference 2011 (SICE2011)* (Tokyo: The Society of Instrument and Control Engineers (SICE) and IEEE), 2178–2181.

Hammer, T., and Bäuml, B. (2014). The communication layer of the aRDx software framework: highly performant and realtime deterministic. *J. Intell. Robot. Syst.* 77, 171–185. doi:10.1007/s10846-014-0095-9

Hayes-Roth, B. (1985). A blackboard architecture for control. *Artif. Intell.* 26, 251–321. doi:10.1016/0004-3702(85)90063-3

Henderson, N., King, R., and Chalup, S. (2008). "An automated colour calibration system using multivariate Gaussian mixtures to segment HSI colour space," in *Proceedings of the 2008 Australasian Conference on Robotics and Automation (ACRA 2008)*, eds J. Kim and R. Mahony (Australian Robotics & Automation Association).

Houliston, T., Metcalfe, M., and Chalup, S. K. (2015). "A fast method for adapting lookup tables applied to changes in lighting colour," in *RoboCup 2015: Robot World Cup XIX, Volume (9513) of Lecture Notes in Artificial Intelligence (LNAI)* (Berlin; Heidelberg: Springer), 190–201.

Kalakrishnan, M., Chitta, S., Theodorou, E., Pastor, P., and Schaal, S. (2011). "Stomp: stochastic trajectory optimization for motion planning," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on* (IEEE), 4569–4574.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). "RoboCup: the robot world cup initiative," in *Proceedings of the First International Conference on Autonomous Agents, AGENTS '97* (New York: ACM), 340–347.

Kulk, J., and Welsh, J. S. (2012). "A NUPlatform for software on articulated mobile robots," in *Leveraging Applications of Formal Methods, Verification, and Validation, Communications in Computer and Information Science*, eds R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen (Berlin; Heidelberg: Springer), 31–45.

Lim, J., Shim, I., Sim, O., Kim, I., Lee, J., and Oh, J.-H. (2015). "Robotic software system for the disaster circumstances: system of team KAIST in the DARPA robotics challenge finals," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)* (Danvers: IEEE), 1161–1166. doi:10.1109/HUMANOIDS.2015.7363509

Magyar, G., Sinčák, P., and Krizsán, Z. (2015). "Comparison study of robotic middleware for robotic applications," in *Emergent Trends in Robotics and Intelligent Systems, Volume 316 of Advances in Intelligent Systems and Computing*, eds P. Sinčák, P. Hartono, M. Virčíková, J. Vaščák, and R. Jakša (Cham: Springer International Publishing AG), 121–128.

Matamoros, J. M., Savage-Carmona, J., and Ortega-Arjona, J. L. (2015). "A comparison of two software architectures for general purpose mobile service robots," in *2015 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC 2015)*, eds A. Valente, R. Morais, L. Almeida, and L. Marques (Los Alamitos: IEEE Computer Society), 131–136.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 43–48. doi:10.5772/5761

Murch, C. L., and Chalup, S. K. (2004). "Combining edge detection and colour segmentation in the four-legged league," in *Proceedings of the 2004 Australasian Conference on Robotics & Automation (ACRA'2004)*, eds N. Barnes and D. Austin (Australian Robotics & Automation Association).

Orebäck, A., and Christensen, H. I. (2003). Evaluation of architectures for mobile robotics. *Auton. Robots* 14, 33–49. doi:10.1023/A:1020975419546

Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*, 2nd Edn. (Upper Saddle River, NJ: Yourdon Press).

Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach*, 6th Edn. (Columbus, OH: McGraw-Hill Education).

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *2009 IEEE ICRA Workshop on Open Source Software*, Vol. 3.

Quinlan, M. J., Chalup, S. K., and Middleton, R. H. (2004). "Application of SVMs for colour classification and collision detection with AIBO robots," in *Advances of Neural Information Processing Systems (NIPS'2003)*, Vol. 16, eds S. Thrun, L. Saul, and B. Schölkopf (Cambridge, MA: The MIT Press), 635–642.

Röfer, T., Laue, T., Müller, J., Burchardt, A., Damrose, E., Fabisch, A., et al. (2011). *B-Human Team Report and Code Release 2011*. Available at: http://www.b-human.de/downloads/coderelease2012.pdf

Thórisson, K. R., List, T., DiPirro, J., and Pennock, C. (2005a). *A Framework for AI Integration*. Technical Report, RUTR-CS05001. Reykjavik University Department of Computer Science.

Thórisson, K. R., List, T., Pennock, C. C., Dipirro, J., Magnusson, F., Thórisson, K. R., et al. (2005b). "Whiteboards: scheduling blackboards for interactive robots," in *AAAI-05 Workshop on Modular Construction of Human-Like Intelligence, Twentieth Annual Conference on Artificial Intelligence, Pittsburgh, PA, July 9-13, 2015* (Palo Alto, CA: AAAI).

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.