The TRADE Middleware for Advanced Robotic Architectures

Matthias Scheutz

Human-Robot Interaction Laboratory
Tufts University
177 College Avenue
Medford, MA 02155 USA
matthias.scheutz@tufts.edu

Abstract

Over the last decade, the Robot Operating System (ROS) has become the *de facto* standard for robotic middleware having significantly improved some of the shortcomings of version 1 with the release of version 2. Yet, while the focus of ROS 2 has been "downward" on the underlying communication layer, the interfaces "upward" to the robotic architectures implemented in ROS have received little attention. In this paper, we argue that robotic middleware can serve important roles for robotic architectures, in particular, cognitive robotic architectures, if the right kinds of interfaces are provided that allow for a tight integration between architecture and middleware. We introduce the *Thinking Robots Agent Development Environment*, TRADE, which is an extension of the previous *Agent Development Environment*, ADE, and provides advanced features for architecture integration and interactions between cognitive robotic architecture and the middleware layer. We describe several features in TRADE that are missing in ROS, in particular, system-wide locking mechanisms, service instrumentation, and middleware service calls and discuss how they can support the architecture developers with implementing advanced architectural features such as dialogue-based system debugging and configuration, or multi-effector multi-robot behavior coordination.

Introduction

Robot middleware was introduced in the early 2000s to provide a layer of abstraction between a computer's host operating system and a robot's control architecture to facilitate the implementation of robot architectures. While middleware systems differed with respect to the range of supported functionalities, they at the very least provided methods for starting architectural components on different devices and enabling communication among the distributed components, sometimes providing a unified messaging format. Most middleware systems also provided additional functionalities borrowed from agent-oriented programming, typically found in early multi-agent systems such as RETSINA (Sycara et al. 2003), for example, agent discovery services, service registries, service brokering, etc. Some middleware systems provided more robot-specific functionality such as streaming data, localization and mapping services, and mechanisms for fault-tolerant computing (e.g., see (Kramer and Scheutz 2007) or (Elkady and Sobh 2012) for an overview of early robot middleware).

Over the years, the Robot Operating System (ROS) (Macenski et al. 2022; Quigley et al. 2009) hosted by the Open Source Robotics Foundation (see https://www.openrobotics.org/) has become the *de facto* standard middleware for robotic systems with broad support for different robots and, most importantly, a large number of implemented components that provide core robot functionality such as simultaneous localization and mapping (SLAM), motion and task planning, and many others, together with a set of visualization, logging and debugging tools that aid robot architecture and application development (e.g., see https://www.ros.org/). While the large ecosystem provided by ROS had important effects on lowering the entry threshold for robotics research, it also had the side effect of stifling the research and development of new middleware that provided better and more integrated functionality for cognitive robotic architectures.

The aim of this paper is to introduce such a new middleware system called TRADE (for "Thinking Robots Agent Development Environments") that is based on and extends core concepts of the *Agent Development Environment* (Scheutz 2006) to provide dynamic distributed fault-tolerant real-time computation across heterogeneous networks of devices and operating systems. Different from ROS, TRADE can run on any operating system with a Java virtual machine and allows for networked architectures across different types of open and secured networks (including firewalls). Moreover, unlike ROS, it provides mechanisms for deep introspection and integration with cognitive robotic architectures as well as mechanisms for fault detection and fault recovery which are critical for the long-term autonomous operations of robots.

The rest of the paper is organized as follows. We start with a brief motivation of middleware features that are necessary for enabling a deep integration with cognitive robotic architectures, followed by a brief feature comparison between TRADE and ROS. We then hone in on some of the unique features in TRADE and briefly describe how they can be utilized in advanced robotic architectures. We conclude with an outlook for robot middleware developments in the age of foundation models.

Motivation

There are different types of robotic middleware, ranging from mere communication infrastructure (e.g., DDS) which is a transparent layer of abstraction, to deep integration between middleware and architecture as is the case with TRADE. ROS 2, for example, uses DDS to address "four features that were lacking in ROS 1 that are directly addressed in ROS 2 through the use of DDS. These four features are: reliability, robustness, security, and support for multi-robot systems." (Portugal et al. 2025). ROS 2 then provides functionality on top of DDS to implement different methods for consuming services, and a software ecosystem with visualization, simulation tools, and implemented components/services like the "MoveIt" motion planning framework that can be used in robotic architectures.

While the transition from ROS 1 to ROS 2 was specifically prompted by the need for "security, reliability in non-traditional environment, and support for large scale embedded system" (Macenski et al. 2022), i.e., a focus on the underlying communication infrastructure to gain real-time support and multi-robot communication as required by many industrial applications, improvements to the upward interfaces to cognitive robotic architectures were largely ignored. For example, ROS does not provide "middleware hooks" that a cognitive architecture could utilize for dynamic modifications and adaptation of the instantiated architectural components. Such hooks could, for example, be used to address issues like fault detection and recovery, or to support opportunistic planning, or for allowing natural language debugging dialogues about the architecture and middleware configuration and operation. But to enable such functionality, proper middleware tasks such as making connections between middleware components, instantiating or removing components, etc. would need to become first-class services accessible in robotic architectures, a feature not present in any robotic middleware other than TRADE to our knowledge.

In general, once the limiting view of robotic middleware as sole communication infrastructure is replaced by a wider perspective of the synergistic potential of middleware and architecture, various middleware functions can be isolated that would provide extended support for cognitive architectures and significantly improve their operation and thus the long-term autonomy of robot systems. Specifically, robotic middleware can

- allow the implemented architecture to control as well as automatically and adaptively distribute its architectural components based on its needs (e.g., to allow for parallelization, faster execution, load balancing, real-time computation, etc.)
- abstract over all networking details to allow architecture programmers to treat the distributed system like one monolithic architecture with components executing in parallel (e.g., call any service from anywhere without the need to subscribe to services, operate on clusters of services or sequences, provide access control and locking at the service level, etc.)
- provide the implemented architecture with operating system and device-independent event notifications and functionality discovery which are particularly important in robot applications with little to no human intervention (e.g., can be used to detect new capabilities dynamically, or faulty ones that can no longer be used)
- provide methods for multi-robot collaboration and coordination (e.g., methods for advanced fault detection utilizing devices on different robots such as cameras to investigate faults) and overall task-based adaptive system configuration in response to changes in the underlying computational infrastructure as well as task environment
- provide introspective access to all middleware system states and processes that the architecture can use to enable natural language dialogues about the middleware, system configuration, etc. (some version of Westworld's "analysis mode", e.g., https://westworld.fandom.com/wiki/Analysis_Mode)

The above is by no means an exhaustive list, but can serve as a starting point for what types of interactions robotic middleware could have with a robotic architecture implemented in it. We will next review how the above features were partly available in the Agent Development Environment (Scheutz 2006) and paved the way for the development of its second generation version, the *Thinking Robots Agent Development Environments* (TRADE).

The Agent Development Environment (ADE)

ADE (Scheutz 2006) was an agent-based middleware implemented in Java to provide broad coverage across different operating systems and virtual machines, using Java RMI for interprocess communication.[1] It had mechanisms for the automatic configuration and startup of distributed architectural components, with system-wide logging capabilities and multi-session GUIs for administrating and monitoring distributed systems. ADE also added access control and security mechanisms to the middleware—a first at that time which to this day has not systematically been integrated into robotic middleware—allowing for the secure execution of distributed architecture components on possibly untrusted hosts. ADE also enabled "component-sharing" across multiple architectural instances, for example, it allowed to share the vision component using a special cellular neural network chip with multiple robots (e.g., (McRaven et al. 2004)). ADE also utilized mechanisms for architectural introspection for fault detection and systematic error recovery (e.g., for automatic restart of crashed components during human-robot interactions (Kramer and Scheutz 2006) or relocation of components to new hosts based on recovery policies if the host is unavailable).

Despite introducing several useful novel features, the design of ADE imposed several limitations on how it could be

[1] ADE started as the "APOC development environment" in the early 2000s specifically geared towards providing the implementation infrastructure for the APOC architecture (Andronache and Scheutz 2004) which later became the foundation for the DIARC architecture (Schermerhorn et al. 2006). The latest public release of ADE is available at https://ade.sourceforge.net/.

used for robotic systems. For one, it had a *centralized registry* (akin to the ROS 1 master) that provided naming and discovery services as well as established connections among components which were required for service calls from one component to another using a special handle. While this facilitated startup and discovery, the central registry also presented a bottleneck in setting up the system (e.g., it had to be directly accessible by all components) and the requirement to explicitly request a connection to another component before its services could be consumed was often cumbersome and led to programming errors. Furthermore, Java RMI was used for remote method invocation together with Java serialization which was convenient at that time as a better RPC but could not bridge subnets and private/public networks, and also imposed limitations on communication and interconnectivity with other systems (not to mention inherited Java RMI problems, e.g., with the Java RMI registry). ADE also only allowed one "ADE component" to be instantiated in a single JVM (i.e., one OS process) in order to be able to introspect on the instantiated component and recover it upon crashes which made it impossible to define "derived ADE components" from other Java objects (ADE components, however, could be used to derive other components). Finally, an ADE system required all class definitions to be available for all components on all JVMs on all hosts even if those classes were never used on those hosts which was a frequent source of "missing class definition" errors.

To address most of these limitations while keeping the overall functionality in place, we developed TRADE, the next generation of ADE, intended as an industry-strength middleware for the long-term operation of intelligent robotic systems.

The Thinking Robots Agent Development Environment (TRADE)

TRADE is a lightweight flexible reflective service-oriented middleware for multi-agent systems based on ADE. It consists of any number of *TRADE containers* which can host any number of *service providers* that offer *services* to any consumer across a heterogeneous socket-based communication network on heterogeneous operating systems and hardware.[2] When TRADE does not need socket communication (across processes or hosts), it uses direct Java method invocation instead (in strict mode, with copies of all arguments) to speed up service calls. TRADE also uses libraries to provide access to C libraries (via JNI) and Python (e.g., https://github.com/mscheutz/diarc/wiki/PyTRADE) as well as ROS systems via special wrapper nodes and can thus utilize any ROS 1 or ROS 2 component within the same TRADE system, together with non-ROS components.

[2]While communication across sockets is currently still based on Java serialization, this will be replaced in the future with more general serialization methods (e.g., ASN.1 or protobuf) to allow for the implementation of native, non-Java TRADE containers in other programming VMs (e.g., Python).

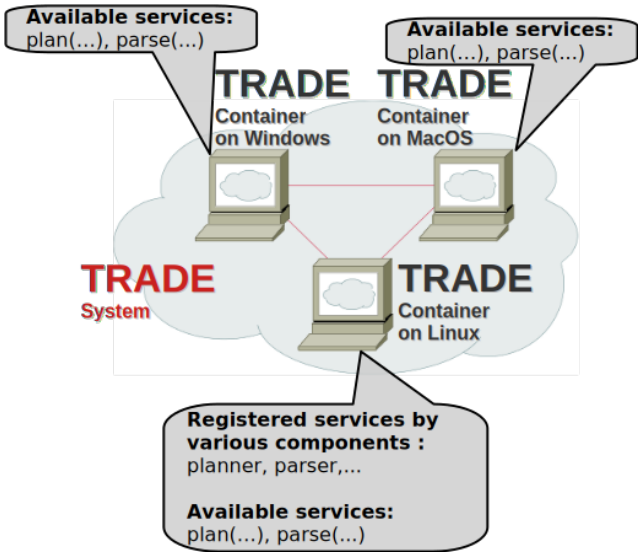

Figure 1: Example of a fully connected TRADE system (red lines) with three containers each running on separate host computers with a different operating systems. The Linux container has planner and parser components among others which registered their respective "plan" and "parse" services, thus making those services available for consumption in all containers.

TRADE Containers

TRADE containers are automatically instantiated in a JVM whenever a TRADE system call is made (e.g., an attempt to look up a service). Once TRADE is launched, it will attempt to automatically discover other TRADE containers and connect to them through sockets if they each have the right credentials and belong to the same TRADE system, potentially forming a fully connected graph depending on their configuration and the underlying network topology (see Fig. 1).

Specifically, TRADE containers can be connected in multiple ways across different public and private networks with hosts running different operating systems, e.g., through automatic discovery but also by explicitly pointing containers to IP addresses and ports (multiple IP addresses can be used for connections, allowing containers to bridge public/private subnets and forward messages between containers that do not have direct socket connections). TRADE containers host service providers and potentially consumers (i.e., Java objects that have @TRADEService annotations) which make their services available system-wide within TRADE with access control upon registration (deregistration removes the services system-wide).

TRADE Services

A *service* in TRADE is any `public` Java method with a `@TRADEService` annotation, either in a Java class or in a Java interface which makes the method in the class (and all implementing methods of interfaces) *eligible* in the TRADE system. Services can be made available and used anywhere anytime in a *TRADE system* but also altered or removed any-

time. They can be chained by defining other services to be called before or after a service, and they can be locked so that only the owner of the lock can call them (chaining and locking are advanced operations that will be discussed later). All services in TRADE are fully introspectible in the system, i.e., information about their argument signatures, return values, resource requirements, access requirements, etc. can be obtained anywhere in the TRADE system. Similarly, all services can be consumed from any place in the TRADE system provided the consumer has the appropriate access permissions. Different from ADE, service calls do not use Java RMI any longer, hence do not have to implement the Java Remote interface. Services in TRADE can also be used by any Java object—registration of the object is not required, nor does the object have to provide services itself.

To be *available* (and consumable) in the system, services must be *registered*. Suppose we have defined the following class `RobotArm` that implements various actions for a robotic arm:

```
public class RobotArm {
  ...
   // open the gripper at given speed
  @TRADEService
  public boolean openGripper(int speed) {
    ...
  }
  ...
}
```

The RobotArm arm class consists of a method `openGripper` which can be registered and thus made available system-wide:

```
  ...
  RobotArm arm = new RootArm(...);
  TRADE.register(arm,null);
  ...
```

Here, `TRADE.register` takes an object and a `String` (or a `Collection<String>`) which is the name of an (optional) *group* to which all methods annotated with `@TRADEService` should belong (see below). In this case, no group names are supplied and `openGripper` is thus made available system-wide without any specific group labels. But when services logically belong together, they can be *grouped* which later allows other services and processes to discover the services they need to perform their operations based on their groups. For example, we could have added the "MyRobot" group identifier to the registration to indicate that the registered method belonged to group called "MyRobot", possibly with other methods from different registered Java objects.

```
  ...
  RobotArm arm = new RootArm(...);
  TRADE.register(arm,"MyRobot");
  ...
```

In the above case, for example, a vision component with a "checkFor(String objecttype)" could also be registered and added to the "MyRobot" group to indicate that the service uses the camera on the particular robot arm (even though the vision code is implemented in a separate class). This allows users to find the right set of services, e.g., when there are more arms and cameras available in the system.

Groups in TRADE can be added and modified at run-time and used either directly or connected to identifiers that are meaningful to an agent. For example, a multi-agent planning algorithm generating an action for a particular robot will need an identifier to make reference to that robot in the plan and that identifier needs to be linked to the actual physical hardware on which the action is supposed to be executed. Groups make this link easy by allowing processes to discover services at run-time, for example all methods available for "MyRobot":

```
  ...
  Collection<TRADEServiceInfo>
    myRobotServices =
      TRADE.getAvailableServices(
         new TRADEServiceConstraints()
               .inGroups("MyRobot"));
  ...
```

Here, the variable `myRobotServices` will consist of a set of returned *handles* of the type `TRADEServiceInfo` which can be used to consume those services if the user has the appropriate access privileges. The `TRADEServiceInfo` contains information about the service name, the service arguments and the return type, and the groups it belongs to which user processes can obtain. The above `TRADE.getAvailableServices` method allows for defining search constraints in terms of `TRADEServiceConstraints` to select particular services (if no constraints are given, handles for all available services are returned).

Suppose we want to use the `openGripper` service:

```
  ...
  for(TRADEServiceInfo s:myRobotServices) {
    if (s.serviceString
           .equals("openGripper(int)") {
      try {
        // open the gripper with speed=3
        s.call(void.class,3);
      } catch(TRADEException te) {
        // handle the failure
        ...
      }
    }
  }
  ...
```

Note that we iterate through the various services associated with the "MyRobot" group to find the one we want and then call it with the class of its return type (in this case "void.class" as it does not return any values) and its "integer" speed argument. The call must be enclosed in a "try-catch statement" to be able to react to different call failures indicated by a *TRADEException*. If calls have to be made more than once, then it is advisable to save the handle and simply reuse it anywhere in the system, e.g.,

```
  ...
  TRADEServiceInfo myGripper;
  for(TRADEServiceInfo s : myrobotservices) {
    if (s.serviceString
```

```
        .equals("openGripper(int)") {
    myGripper = s;
    break;
  }
}
try {
  myGripper.call(void.class,3)
} ...
```

In general, services can either be called by selecting them on-the-fly via *service constraints*, but it is also possible to obtain a token (handle) to a desired service (again via a selection process) that can subsequently be passed around in the system and used for calling the service at any time while the service is available (a call to a previously registered but subsequently deregistered service will fail). TRADE allows for different ways to discover available services, e.g., searching by service name and constraints, but also *service semantics* (e.g., post-conditions) if they are provided in the service definition. Constraints are conjunctions ("and") in disjunctive normal form ("or") with negation ("not") using, for example, the following keywords: "local" (the service runs in the same JVM as the consumer), "host" (the service runs on the specified host), or "group" (the service is in the specified group of consumers). For example, if it is known that there is only one service (with no arguments) available in the "MyRobot" group that is running on the same host (but possibly a different TRADE container) as the consumer, the service can be called without making reference to its name:

```
TRADE.getAvailableServices(
       new TRADEServiceConstraints()
          .inGroups("MyRobot")
          .local()).call(void.class);
```

Novel Features in TRADE

While robotic middleware is typically always in active development and new features are added all the time, it is still useful to compare the current version of TRADE with ROS 1/2 to see what distinguishing features TRADE can offer that are not present in ROS but also where TRADE is lacking direct support (see the brief overview in the table below).

ROS 1/2	**TRADE**
Service (call from within nodes), blocking vs. non-blocking	Service (call from anywhere in the JVM) non-blocking via "once" notifications
Actions	Service/Notifications and Actions in DIARC
Pub/sub	Notifications
Introspection into services (names, arguments	System-wide introspection
Discovery in DDS (nodes, topics)	Auto-discovery and configuration
Peer-to-peer connections necessary	Heterogeneous network (forwarding)
Callbacks	Notifications
	before/after calls
	service lock
	middleware methods as service calls

ROS 2 made significant improvements over ROS 1 with respect to service calls, where they can occur (essentially anywhere within a ROS node) and whether they are blocking or non-blocking. Similarly, TRADE allows for service calls from anywhere in the JVM. While all calls are blocking, non-blocking calls can be made via "once" notifications (i.e., scheduling a call-back to the place where the non-blocking call ought to continue in the code). ROS actions have equivalent mechanisms in services and notifications in TRADE, and, to a much more extensive way, in the "Action system" implemented in DIARC (Scheutz et al. 2019). While the ROS pub/sub mechanism does not directly exist in TRADE—all calls are connection-oriented—extensive notification mechanisms in TRADE allow for the same behavior as well as more advanced conditional streaming methods (albeit at the cost of reduced transfer speeds). Introspection within ROS nodes corresponds to system-wide introspection in TRADE, and ROS callbacks can again be accomplished via notifications in TRADE. DDS discovery mechanisms are covered by auto-discovery mixed with potential system configurations in TRADE, and while peer-to-peer connections are necessary in ROS, TRADE natively supports forwarding of message across containers in case fully connected container graphs are not possible (e.g., in cases of mixed private/public networks with firewalls). In addition, TRADE supports "before/after" calls, which are ways to instrument service calls at run-time, as well as a middleware service calls, neither of which exist in ROS. We will briefly discuss these unique features after a short summary of the TRADE notification system.

TRADE Notifications

TRADE has a rich notification system that can be used with callbacks to interested consumers under various conditions and for different purposes that are particularly useful for cognitive architectures. For example, "joined" and "left" notifications are used to inform interested parties whether services became available or unavailable system-wide. In a distributed cognitive system where components might not be available all the time (e.g., a knowledge base might be down, or a new knowledge base might become available), these notifications allow components to receive updates on the status of services they need to use (e.g., if a knowledge base is unavailable, the system might use a different method for continuing its task; or if a new knowledge base becomes available, the system might decide to use it together with existing knowledge bases or instead). Here is an example from the DIARC architecture's "Reference Resolution" component registering a notification to be called back via "newConsultantCallback" when a new (consultant) component with a "getKBName" service joins :

```
...
  try {
    TRADE.requestNotification(this,
       "joined",
       new TRADEServiceConstraints().
                 name("getKBName"),
       null, "newConsultantCallback");
  } catch (TRADEException e) {
```

```
    log.error("Notification failed");
    ...
  }
...
@TRADEService
  public void newConsultantCallback(
             TRADEServiceInfo tsi) {
  ...
}
```

Another important notification, the "once" notification, can used to implement *non-blocking calls*, e.g., in cases where a service might take a long time (such as an involved task planning process or a "mental simulation" in the cognitive architecture). They can also be triggered "after" a particular interval, or scheduled "at" a particular time, or at "every" scheduled interval time (e.g., for polling the state of other components or checking on whether a condition in another component is met). In all cases, callbacks need to be provided which themselves are service calls and can thus be dynamically scheduled from anywhere in the system and also be canceled anytime from anywhere (this is different from ROS which does not provide service-oriented callbacks). This means that components requesting a notification (e.g., the goal manager in a cognitive architecture) do not have to be the ones providing the callback service (e.g., the goal manager could schedule an "every" notification to regularly check whether "isLowBattery()" is true and then automatically issue a service callback to the "dock" service). Overall, notifications can be used to implement rich event-driven interactions beyond pub/sub mechanisms that reduce computational overhead associated with checking for results. For example, an "if" conditional notification will only trigger a callback if a condition is met at the time the callback is scheduled, while a "when" conditional notification will remain in effect until the condition is met, at which time the callback is initiated and the notification is canceled. By allowing individual architectural components to schedule notifications that trigger service calls (under particular conditions), TRADE supports a new way of streamlining event-based interactions among components in a distributed cognitive architecture.

TRADE Before/After Calls

A novel features, inspired by aspect-based computing and not available in other middleware systems is the online instrumentation of service calls with other services calls. TRADE supports so-called "before/after" service calls from anywhere in the TRADE system which can be used to dynamically adapt and modify the behavior of existing service calls (note that this is a significant generalization from instrumentation methods that are limited to the VM in which a service is called). For example, whenever a service S is called, the "before" service call B is called with S and its arguments, and the "after" service A is called like B with the additional return value from S, i.e., for $S(arg_1, \ldots, arg_n)$ we get the following calling picture:

call$(B, S, arg_1, \ldots, arg_n)$
ret = call$(S, arg_1, \ldots, arg_n)$
call$(A, S, arg_1, \ldots, arg_n, ret)$
return ret

Note that A and B can themselves have "before/after" calls (which makes it important to avoid circular call invocations), and that the return value from S ("ret") is returned at the end of the call sequnce and that "before/after calls" cannot alter the invocation of S or its return value (to ensure that the original messages are passed on).

```
...
 @TRADEService
 public void addObserver(
         String serviceName,
         String[] serviceArgsClasses,
         TRADEServiceConstraints constr) {
     try {
         TRADE.afterService(
            "addObservedOutput",
            new String[]{Object[]
                       .class.getName()},
            new TRADEServiceConstraint(),
            serviceName,
            serviceArgsClasses,
            constr);
     } catch (TRADEException e) {
       ...
     }
  }
```

Here, "addObserver" uses a TRADE after-service call to add an "addObservedOuput" service which, for example, could log return values from the service that was passed in through "serviceName". While observers are one application of instrumenting service calls, they can also be used in many other ways, for example, to introspect on service call patterns in a running system that can be used to learn an internal model of the system's operation (cp. (Berzan and Scheutz 2012) where logs of the service calls in ADE were used to learn a system model offline). Such a learned system model can be useful for complex architectures (where it is practically impossible to build a system model by hand) or for learning systems that change their behavior over time to detect deviations from previously learned models. The dynamic nature of call instrumention is particularly interesting for cognitive architectures that use attention mechanisms for determining which internal states to attend to at different times (those could be obtained through attaching and detaching before/after service calls). Before and after calls can also be used to implement updates of agent activities in multi-agent systems (e.g., to inform an agent whenever other agents perform particular actions).

TRADE Service Locks

Behavior arbitration and how to ensure mutually exclusive access to different robot effectors, especially in distributed systems, is an ongoing challenge in robotic architectures. And usually, it is the robot programmer who has to ensure that only a single process can access a single effector at a time. When multiple effectors are needed to perform a particular behavior (such as walking in a humanoid robot where

both legs need to be controlled by the gait process) it is particularly important to "lock down" all involved effectors and controllable joints so that no other process can incidentally interfere with the control process. For this and other purposes, TRADE provides a novel distributed locking mechanism that operates at the level of services rather than the level of programs, scripts, devices or other resources. Services, as first class objects in TRADE, are accessible anywhere in the system, hence with the novel locking mechanism they can also be locked and unlocked together from anywhere in the system (note that this is very different from having an OS-based lock that will only work on the host system). A process requiring access to multiple services (possibly being offered in different containers on different hosts) can simply request a lock, and if the lock to all requested services can be obtained because no other process holds a lock on one of them, the requesting process will get the lock and can subsequently use these services exclusively without any interferences from other processes. Once the critical operation has concluded, the services can be collectively unlocked and are then available again for use by other processes. For example, a target finding process might need to turn a humanoid robot's head with the embedded cameras to be able to look around while the navigation process also needs control of the head in order to focus the camera in the movement direction to be able to spot obstacles. In that case, the target finding process might either have to wait for the navigation process to release the locks on the head or it could use a mechanism like the one proposed in (Krause, Schermerhorn, and Scheutz 2012) to make a request for the robot's action execution system to stop walking, at which point the navigation process would automatically release the lock.

TRADE Middleware Service Calls

Another unique feature in TRADE (not available in ROS) is the reflective access to TRADE system services such as broadcasting a TRADE container's system information or establishing a connection between containers, local and remote registration and deregistration of services, etc. all of which enable architectures to reason about the system configuration and performance, and to potentially plan system-type operations that improve the architecture's operation and performance. For example, suppose a task planner needs access to a vision device in order to make an observation in the environment, but the current system does not include one. The planner could then plan to turn on TRADE's "auto-discovery" to see if any potential TRADE containers with vision services are available that it could connect to. If it has the right credentials, the planner's container can then join the other container, and if that container is part of a system that has a component with a vision service in one of its containers, the service will be discovered and made available to the planner for use.

Another example of using middleware service calls would be to enable "configuration debugging" in natural language. In this case, the language system would allow for dialogues about existing services, where they are hosted, what containers exist in the system and where, etc. All that is needed is to enable natural language interactions about the system configuration in a cognitive system would be to connect the natural language terms in its lexicon to the respective system calls and to generate again natural language expressions from the return values of these calls. For example, the natural language instruction "Turn on discovery" would be translated into "TRADE.setDiscovery(credentials,true)" (where "credentials" are the speakers access credentials stored somewhere in the architecture) which would turn on scanning for new accessible TRADE containers on the network and subsequent connections to them. A follow-up could then ask for the newly available services which would require some internal filtering of old services on the list obtained from "TRADE.getAvailableServices()".

In addition, middleware service calls allow for the definition of policies for how to handle different system configurations in different contexts (and these policies could potentially even be learned). For example, a policy my specify how to adapt a TRADE system based on the availability of resources, CPU load, available devices, etc. New service providers could be instantiated and registered at a remote host to provide new services, services could be migrated to other hosts, services could be terminated, etc., all based on other types of contingencies defined in the system's cognitive architecture. For example, a simple policy to bootstrap a system might be to "turn on broadcast" (a middleware function that initiates container discovery by broadcasting a container's information), wait for a particular service to become available, then turn off broadcast. This policy could, for example, be used by the planner in the example above to find the vision service it needs to execute its plan (e.g., because one of the planning actions has as a precondition "see(object").

It is also possible to use system calls to remotely start a container on a particular host (assuming the host is reachable on the network and the executing container has access rights through a secure shell call), and then make it instantiate a particular service provider which registers its services that will then become available system-wide (again, this could be the vision service provider being instantiated on a host with a camera device).

Discussion and Conclusion

A major aim of this paper was to demonstrate that middleware development is not a matter of the past, but that there is room for robotic middleware to grow and provide ever better and tighter integration with cognitive robotic architectures that, in turn, can enable improved performance of robotic systems. In this sense, TRADE can serve as an example of how robotic middleware can take stock of existing features and aim to implement desiderata that will advance and improve robotic architecture development. Specifically compared to the functionality available in TRADE's precursor ADE, TRADE is only missing the automatic restarting feature for crashed components which is not fully possible given constraints on Java introspection (even in ADE, which was specifically designed to enable this feature, full recovery required additional steps such as saving component state at regular intervals to be able to restore it). At the same time, the added flexibility of TRADE allows for future

non-native Java TRADE containers, more efficient multi-language communication protocols (e.g., (Liu et al. 2022)), additional call methods like pub/sub for efficient streaming, flexible distributed multi-language, multi-OS systems, and mixed middleware architectures (such as bridging ROS 1 and 2 systems which is already possible; past middleware integration in ADE such as JAUS could also be enabled). Compared to ROS, the advanced introspection and notification as well as middleware service call features allow for the development of more introspective fault tolerant robotic systems where the cognitive robotic architecture, due to its tight integration with TRADE, can take explicit steps for handling interruptions and perturbations, potentially even learning its own best recovery policies over time. The aspect-oriented "before/after" instrumentation enables a large number of dynamic architectural extensions, from adding or removing explicit debugging information at run-time, to system-wide online profiling of call sequences, to explicit user notifications when particular conditions obtain, and many more. The advanced system-wide service-based locking mechanisms enable novel ways for robotic architectures to explicitly and dynamically manage groups of services that logically belong together and should be called in a particular coordinated manner. Critically, the service-based locking mechanisms removes the otherwise cumbersome programming of distributed locking mechanisms which are difficult to implement across hosts and operating systems and are generally error-prone. By allowing to lock groups of services at a time, it provides locks at the right level of abstraction for the architecture developer who can view the distributed system as one integrated platform, very much in the spirit of the original monolithic cognitive architectures. And by providing introspection into the owner of locks and the usage status of locked services, TRADE allows for the realization of sophisticated adaptive behavior arbitration schemes across parts or the whole TRADE system. Like ROS, TRADE is being actively developed, albeit by a small group of developers. Nevertheless, its capabilities might already prove useful to the larger robotics community. It is thus freely available for non-commercial use as part of the public DIARC repository at https://github.com/mscheutz/diarc/.

References

Andronache, V.; and Scheutz, M. 2004. Integrating Theory and Practice: The Agent Architecture Framework APOC and its Development Environment ADE. In *Proceedings of AAMAS 2004*, 1014–1021. ACM Press.

Berzan, C.; and Scheutz, M. 2012. What am I doing? Automatic Construction of an Agent's State-Transition Diagram through Introspection. In *Proceedings of AAMAS 2012*.

Elkady, A.; and Sobh, T. 2012. Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography. *Journal of Robotics*, 2012.

Kramer, J.; and Scheutz, M. 2006. ADE: A Framework for Robust Complex Robotic Architectures. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4576–4581. Bejing, China.

Kramer, J.; and Scheutz, M. 2007. Robotic Development Environments for Autonomous Mobile Robots: A Survey. *Autonomous Robots*, 22(2): 101–132.

Krause, E.; Schermerhorn, P.; and Scheutz, M. 2012. Crossing Boundaries: Multi-Level Introspection in a Complex Robotic Architecture for Automatic Performance Improvements. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Liu, W.; Jin, J.; Wu, H.; Gong, Y.; Jiang, Z.; and Zhai, J. 2022. Zoro: A robotic middleware combining high performance and high reliability. *Journal of Parallel and Distributed Computing*, 166: 126–138.

Macenski, S.; Foote, T.; Gerkey, B.; Lalancette, C.; and Woodall, W. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66): eabm6074.

McRaven, J.; Scheutz, M.; Cserey, G.; and Porod, W. 2004. Fast Detection and Tracking of Faces in Uncontrolled Environments for Robots Using the CNN-UM. In *Proceedings of CNNA 2004*.

Portugal, D.; Rocha, R.; J.P.; and Castilho. 2025. Inquiring the robot operating system community on the state of adoption of the ROS 2 robotics middleware. *International Journal of Intelligent Robotics and Applications*, 9: 454–479.

Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. 2009. ROS: an open-source robot operating system. In *International Conference on Robotics and Automation (ICRA,) Workshop on Open Source Software*. Kobe, Japan.

Schermerhorn, P.; Kramer, J.; Brick, T.; Anderson, D.; Dingler, A.; and Scheutz, M. 2006. DIARC: A Testbed for Natural Human-Robot Interactions. In *Proceedings of AAAI 2006 Mobile Robot Workshop*.

Scheutz, M. 2006. ADE - Steps Towards a Distributed Development and Runtime Environment for Complex Robotic Agent Architectures. *Applied Artificial Intelligence*, 20(4-5).

Scheutz, M.; Williams, T.; Krause, E.; Oosterveld, B.; Sarathy, V.; and Frasca, T. 2019. An Overview of the Distributed Integrated Affect and Reflection Cognitive DIARC Architecture. In Ferreira, M. I. A.; Sequeira, J. S.; and Ventura, R., eds., *Cognitive Architectures*, Intelligent Systems, Control and Automation: Science and Engineering. Springer International Publishing.

Sycara, K.; Paolucci, M.; van Velsen, M.; and Giampapa, J. A. 2003. The RETSINA MAS Infrastructure. *Journal of Autonomous Agents and Multiagent Systems*, 7(1 & 2).