TeCoLa: A Programming Framework for Dynamic and Heterogeneous Robotic Teams

Manos Koutsoubelias and Spyros Lalis
University of Thessaly & CERTH
Volos, Greece
{emkouts,lalis}@uth.gr

ABSTRACT

Many pervasive computing applications can benefit from the advanced operational capability and diversity of sensing and actuation resources of modern robotic platforms. Even more powerful functionality can be achieved by combining robots with different mobility capabilities, some of which may already be deployed in the area of interest. However, the current programming frameworks make such flexible resource utilization a daunting task, as the application developer is responsible for the laborious and awkward management of the heterogeneity and transient availability of resources and services, which is done in a manual way. TeCoLa is a programming framework addressing the above issues. It supports structured services as first-class entities, which can appear/disappear dynamically, and are accessed in a transparent way. In addition, to ease the task of multi-robot programming, TeCoLa supports the creation and management of teams based on the dynamic service capability and availability of individual robots. Teams are maintained behind the scenes, without any effort from the application programmer. Furthermore, they can be controlled through team-level service interfaces which are instantiated in an automatic way, based on the services of the robots that are members of the team. We present a first implementation of TeCoLa for a Python environment, and discuss how we test the functionality of our prototype using a software-in-the-loop approach.

CCS Concepts

•Applied computing → Service-oriented architectures; •Software and its engineering → Middleware; •Computer systems organization → Robotics; •Computing methodologies → Distributed programming languages;

Keywords

Mobile robots; team programming; high-level coordination; resource dynamics and heterogeneity; pervasive computing;

MOBIQUITOUS '16, November 28-December 01, 2016, Hiroshima, Japan

DOI: http://dx.doi.org/10.1145/2994374.2994397

1. INTRODUCTION

Robotic platforms will play a crucial role in the modern computing landscape, becoming a key component of Cyber-Physical Systems [30] and the so-called Internet of Things. Indeed, a wide range of monitoring and control applications from various domains could be upgraded to a substantial degree or even completely revolutionized using the powerful and growing capabilities of mobile robots. Current robotic platforms not only come at lower cost than ever before but also feature a rich set of on-board computing and sensor/actuator resources. They are also available in many forms and shapes, and are capable of operating and navigating in ground, aquatic and aerial environments [1] [2].

Applications, like precision agriculture [33], forest surveillance [13] and pollution tracking [32], are already beginning to take advantage of single or multiple robots. Putting aside the application-specific particularities, most missions follow a common pattern, namely a given number of robots is used to perform certain sensing and possibly also some actuation tasks in an area of interest. However, the robots used in such missions are typically identical to each other, and their number is pre-defined. This makes missions rather inflexible as it is hard or even impossible to incorporate and exploit new resources at runtime.

We envision applications that are able to exploit a diverse ecosystem of computing, sensing/actuating and mobility capabilities, without being restricted by the initial composition of the robotic group. More specifically, it should be possible to harness additional resources provided by other subsystems, which can become part of the robotic group, on demand, as a function of the application dynamics. These subsystems could be stationary or mobile robots that are already deployed on site, which can be hired in a dynamic way to perform special tasks. Similarly, as this is often the case in human-based operations, additional resources could be brought into the area of interest from remote locations, to increase the overall system capacity or to provide missing capabilities that are needed in order to better observe and handle a detected event, or simply to replace units that fail.

Implementing such functionality can be hard or even impossible using existing programming frameworks. For this reason, we have developed TeCoLa, a programming framework for high-level coordination of robotic teams.TeCoLa provides abstractions and mechanisms to address the above issues of resource heterogeneity and dynamics, allowing the application to engage and exploit robotic services in a flexible way. It comes with novel and unified abstractions for controlling individual robots as well as teams of robots, with

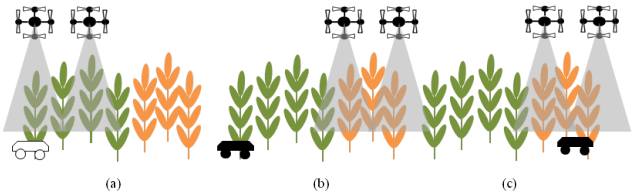

Figure 1: Smart agriculture scenario.

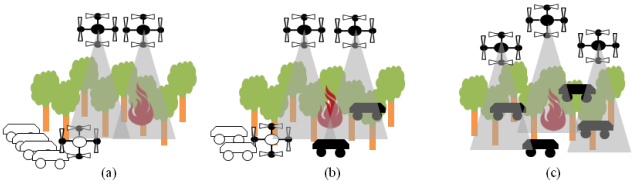

Figure 2: Forest surveillance and fire-fighting scenario.

a minimal extra effort on behalf of the programmer, and with the bulk of team management work being performed behind the scenes. With existing programming frameworks such functionality must be implemented in a manual way.

The rest of the paper is structured as follows. Section 2 presents indicative application scenarios that illustrate the flexibility we wish to support with TeCoLa. Section 3 describes the high-level system model/concepts behind TeCoLa. Section 4 presents a first implementation of TeCoLa for a Python environment, while Section 5 briefly discusses how we test the functionality of our prototype using a software-in-the-loop simulation approach. Section 6 provides an overview of related work. Finally, Section 7 concludes the paper with directions for future work.

2. APPLICATION SCENARIOS

In this Section, we present indicative scenarios for the kind of flexibility we have in mind. These scenarios involve several nodes/robots, which: (i) feature different computing, sensor and actuator resources; (ii) have different mobility capabilities; (iii) are added to and removed from the application in a dynamic way.

Figure 1 illustrates a scenario where a number of unmanned aerial vehicles (drones) is sent out to scan a crop field. Some drones can be equipped with infrared and visible light cameras while others may feature a set of sensors suitable for air quality monitoring. The data collected from these airborne sensors is processed to detect crop deficiencies and take corrective action—for instance, to add extra water/fertilizer or spray some pesticide in the areas of the plantation that are problematic. Since the corresponding machinery can be quite heavy or even dangerous to operate from the air, the field could feature a special unmanned ground vehicle (rover) with the necessary equipment, parked on site (bottom-left corner of Figure 1a). If the drones detect a problem, as shown in Figure 1b, the rover is activated/employed so that it moves to the area of interest and performs the necessary action, see Figure 1c.

As an example with even richer dynamics and collaboration options, Figure 2 shows an application scenario where the objective is to detect and extinguish fires in a forest area. During periods of low humidity and high air temperature where the probability of a fire outbreak is large, the forest is monitored by a group of drones. These are equipped with infrared and visible light cameras, and scan the forest continuously, arranged in a suitable formation that maximizes coverage. At a designated area in the forest there is a station where more drones and rovers equipped with firefighting equipment are parked—these remain inactive as long as no fire incident is reported. When the drones detect a fire, as shown in Figure 2a, the rovers are activated and instructed to move to different positions in order to fight the fire in a coordinated way, see Figure 2b. If the situation remains critical and the fire could get out of control, additional drones and rovers can join the operation to boost the system firefighting capacity, as illustrated in Figure 2c. In a similar vein, some of the commissioned drones and rovers could be withdrawn once the fire is contained.

In order to realize such scenarios, the application must be able to discover and integrate diverse robotic resources, in a flexible and opportunistic way, without making any hard assumptions about their availability and number. On the one hand, it must be possible to extend the system formation by adding new robots, with potentially very different capabilities (e.g., ground vs. air, sensing vs. actuation). On the other hand, it must be possible to adapt system operation to incorporate the newly added robots in a smooth way, with a minimal disruption to the ongoing mission. Similar flexibility is required to discharge and remove robots from the mission, or to deal with failures.

TeCoLa is designed to simplify the programming of such application scenarios. It provides support for resource heterogeneity and dynamics, and offers team-based programming constructs that allow the application to control different robotic teams in a flexible and powerful manner. This way, a large number of mundane and awkward resource access and team maintenance issues, which would otherwise have to be dealt with manually by the programmer, are handled behind the scenes in a transparent way. In turn, this allows the programmer to focus on the essence of the application and mission at hand.

3. SYSTEM MODEL

This section presents the basic concepts and system model behind TeCoLa, for supporting a flexible engagement and exploitation of robots with different capabilities. The concrete programming constructs for supporting this system model are discussed in Section 4.

3.1 Nodes and services

A robot is regarded as a *node* with a set of hardware and software resources. Hardware resources are the node's CPU(s) and other processing elements such as DSPs, GPUs or FPGAs, as well as memories like RAM and Flash. In terms of software, apart from the operating system and drivers for such hardware resources, the node may include various libraries and software components used to perform higher-level data processing tasks.

Based on the resources that are locally available, a node may provide one or more *services* to the outside world. In the spirit of service-oriented design [19] each service is accessed through a structured interface. Without loss of generality, we let service interfaces consist of *methods* and *properties*. Methods are used to invoke the service, while properties are used to expose certain characteristics as well as

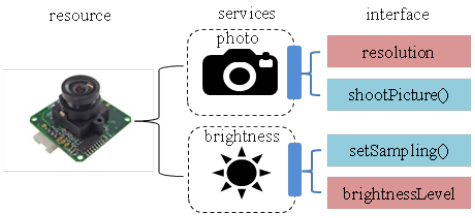

Figure 3: Mapping of resources to services.

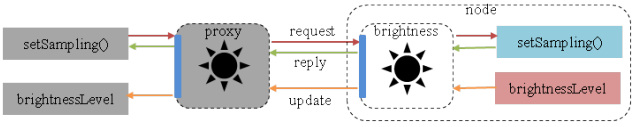

Figure 4: Information flow between a proxy and the node.

state-related information of the service. Note that properties can be dynamic, and change their value as a result of service invocation and/or the internal operation of the node.

As an example, Figure 3 shows a camera resource that maps to two distinct services, with indicative (simplified) interfaces. The photo service interface includes the method of *shootPicture*() for taking a snapshot, and the property *resolution* for the camera's resolution. The brightness service interface consists of the *setSampling*() method for setting the rate at which the camera sensor is sampled, and the *brightnessLevel* property for the last value measured. It is also possible to have a service that relies on more than one resources; as an example, a camera could be combined with an image processing library/component to provide a service for detecting objects of interest.

The node's services can be accessed remotely yet in a transparent way, through a *proxy* that features identical service interfaces, as illustrated in Figure 4. Proxy methods work in the spirit of remote function/procedure calls [11], and block until the node handles the request and returns the reply. The nature of the requested operation can be synchronous or asynchronous. Synchronous operations are performed immediately, and the result can be returned as part of the reply. In contrast, asynchronous operations take longer time to perform, in which case the reply is merely a promise [25] that the requested operation will be eventually performed. The progress of asynchronous operations can be monitored via suitable properties. Note that promises can be broken, due to failures or subsequent method invocations that may cancel previous ones.

Unlike method calls, which map to a bi-directional request-reply interaction, property values flow uni-directionally, from the node to the proxy. Static properties only need to be sent once. Updates on the values of dynamic properties are propagated from the node to the proxy, without any explicit action on behalf of the application.

3.2 Service heterogeneity and dynamics

Nodes can be heterogeneous in their hardware resources. For instance, certain nodes may be equipped with additional or more powerful processing elements and larger memories, while others may feature special sensors/actuators which are too expensive to include in all nodes. Also, nodes can be heterogeneous in terms of software resources. As an example, some nodes may have libraries/components for performing special data processing tasks; these, in turn, may rely on special hardware resources, such as FPGAs.

The aspect of heterogeneity naturally extends to services. For instance, some nodes can support the photo service (based on the camera resource), while other nodes can provide a spotlight service (based on a lamp resource). There can also be heterogeneity among services of the same class. As an example, some nodes may feature higher-resolution cameras or faster and more accurate image processing code than others. The different service classes and interfaces that are relevant for the application domain in question are formally defined using a suitable taxonomy/ontology [10]. Developers must consult this ontology in order to craft their applications so that they can exploit this service variety. The design and human- and machine-readable representation of such a service ontology is beyond the scope of this paper; this front has already been researched extensively in the context of ubiquitous computing systems [16, 21].

Note that the services provided by a node might vary in time. This can be due to dynamic resource availability or local resource management decisions. For instance, if the camera is damaged, the node will lose the ability to support the photo service. A node may also decide to power-down a certain resource to save energy, e.g., the lamp, in which case it can no longer provide the services that depend on it, e.g., the spotlight service.

3.3 Node mobility as a service

We assume that nodes can move in space, in relative terms as well as with reference to a common positioning system such as GPS. Also, nodes come with basic autopilot capability and can deal with low-level motor/control and obstacle avoidance issues in an autonomous way, in the spirit of [23]. Thus the application does not have to micro-manage nodes, and can control their movement in a high-level fashion, by specifying target locations or supplying intermediate waypoints, as usual in many robotic systems [3].

For reasons of uniformity, node mobility is also be captured in a service-oriented way. The respective service interface can include methods for setting a destination, several waypoints or instructing the node to move towards a given direction. It can also include methods for setting orientation and speed. In turn, the current position, speed and orientation of the node can be observed through dynamic properties.

Of course, the actual movement capability of a node depends on what the node is built for — moving on the ground, floating on water, going underwater or flying in the air. This is yet another dimension of heterogeneity, which can be captured by introducing different classes of mobility services and suitable interfaces, via the system ontology. As this is the case for every other form of service heterogeneity, here too, the application must be designed to handle/exploit it in a meaningful way.

3.4 Mission group

The control of the heterogeneous nodes is done following a coordinated approach, whereby a distinguished entity (the coordinator) is responsible for monitoring the state of the nodes, analyzing the situation, taking decisions about the actions that need to be performed, and sending out corresponding commands to one or more nodes. The coordinator

can be a (nearby) command-control center, or one of the nodes which acts the leader of the mission and is in regular contact with the (remote) command-control center.

The nodes that participate in a mission form the so-called *mission group*. This group can grow dynamically to include additional nodes in the course of the mission. It can also shrink in case some nodes leave or fail. The formation of the mission group is managed by the coordinator, which decides which types of nodes/services are needed at any point in time to perform the various tasks of the mission, and accordingly invites new nodes to join or allows participating nodes to leave the group.

The coordinator keeps an up-to-date view of the mission group, which includes a proxy for each participating node. It can browse the mission group in order to inspect the available proxies and see the services supported by each one of them. The coordinator controls individual nodes by invoking the methods and/or accessing the properties of the respective proxies. It can also wait on a condition (expression) that refers to services and service properties. For instance, the coordinator can invoke a method for requesting a node to move to a specific position, and then wait (on the position property) until the node reaches that position.

3.5 Teams and service promotion

Although the coordinator can execute missions by controlling nodes individually, this can be quite awkward in practice; especially if several nodes need to perform the same operations. The typical case is if one wishes for a swarm to scan an area in order to detect certain situations or objects of interest, e.g., fire outbreaks in a forest or shipwrecked persons in the sea.

To this end, the coordinator may form and control different *teams* of nodes. A team is created by applying to the mission group a *membership rule* expression, which can refer to specific services and service properties. Only nodes that match this rule become members of the respective team. Teams are maintained automatically, driven by the specific membership rules. For instance, if a node joins the mission group and its services/properties satisfy the membership rule of a team, its proxy is automatically added to that team. Conversely, if a node no longer matches the membership rule of a team, either because it stopped providing a service or a service property changed value, the proxy is removed from the team. Note that a node can belong to different teams at the same time. A team can be empty, if there is no node that currently matches the membership rule; in this case, the team still exists as an entity that can be inspected and manipulated by the coordinator.

As an example, Figure 5 illustrates the evolution of three teams, for nodes with a photo service, a brightness service and a spotlight service, respectively. Initially, the mission group contains node A which provides the photo and brightness service, and node B which only provides the brightness service. Hence the photo team has one member (A) and the brightness team has two members (A and B), whereas the spotlight team is empty. Then, node C with brightness and spotlight services joins the mission group, and as a result becomes a member of the respective teams.

Services common to all members of a team are *promoted* to the team level (service promotion can be viewed in analogy to inheritance via sub-classing in object-oriented models [22], but works in a bottom-up rather than top-down

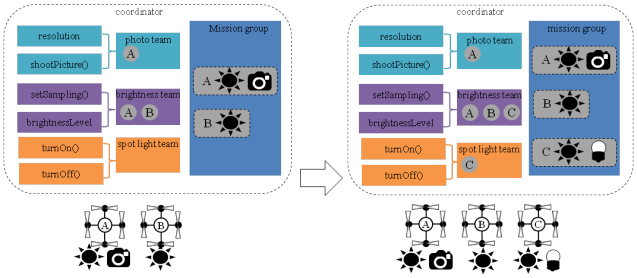

Figure 5: Team dynamics as a function of the current composition of the mission group.

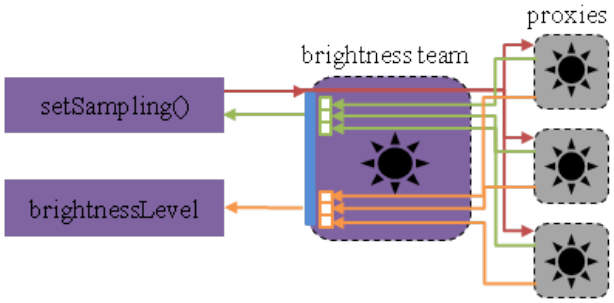

Figure 6: Team-level promotion of service methods and service properties.

way). The coordinator can safely assume that the team will feature the services which are used as a selection criterion in the membership rule. This is indicated in Figure 5 by depicting interface icons next to the team membership lists. Service promotion also applies to any other services that happen to be provided by all team members—however, as this is circumstantial, the coordinator can only exploit such team-level services in an ad-hoc, opportunistic way.

For methods, promotion leads to *identical* methods at the team level, with the same name, arguments and semantics as the methods of the common service interface. The invocation of a team-level method leads to the invocation of the respective method across all proxies. This way, the coordinator can drive the entire team as if it were a single node. Promoted methods return a *vector* of replies, one for each team member. Similarly, for properties, promotion leads to corresponding *vector* properties, with the same name and type. In both cases, the vector size reflects the current team membership. By iterating over these vectors the coordinator can retrieve the reply/property values of individual team members. Figure 6 illustrates in more detail the principle of service promotion for the brightness service team in Figure 5, assuming three team members (proxies).

Last but not least, the coordinator can wait on conditions at the team-level, in practically the same way this is done for individual nodes. The main difference is that in this case the expression refers to promoted team-level services and properties. As an example, the coordinator can instruct a team to move to a certain position, and then wait on a condition with an expression similar to the one used for the movement of an individual node, until all team members have reached their positions.

4. IMPLEMENTATION

This section presents a first implementation of TeCoLa along the lines of the system model described in the previous section. To accelerate prototyping, we introduce this functionality on top of a standard Python environment for a Linux platform, via suitable high-level classes, without touching lower system software layers or making changes to the Python runtime environment.

4.1 Services

Services are implemented as independent software components (singleton classes), derived from the *Service* metaclass. Each service can expose its functionality to the outside world through a number of methods and properties that are accessed via its proxy. As already discussed in Section 3, for all practical purposes, the service class names, methods, properties and semantics, would have to be formally defined via suitable taxonomy/ontology; this will typically be application-domain specific. To give an example, Listing 1 provides a skeleton for a brightness service that offers a simple interface in the spirit of Figure 3 (the *object* class parameter is Python specific). The service also comes with local methods for activating and passivating the service, as will be discussed next.

Listing 1: Skeleton of a service implementation.

```
class BrightnessSvc(object):
  __metaclass__ = Service

  brigthnessLevel

  def setSampling(rate):
    set_sampling_rate(rate)

  def __activate():
    turn_on_brightness_sensor()
    set_sampling_rate(defaultrate)

  def __passivate():
    turn_off_brightness_sensor()
```

Note that a service may internally use threads to implement background or asynchronous activities. For instance, the brightness service could use a thread to periodically read the brightness sensor according to the specified sampling rate, and update the brightness property accordingly. The details depend on the hardware components, drivers and user-level libraries used to implement the desired service functionality. In any case, we assume that services can be developed from scratch by the application developer, or come as pre-installed and ready-to-use components of the robotic platform.

4.2 Node object

A node essentially acts as a service provider. Node-level functionality is encapsulated within an instance of the singleton *Node* class, which is given a type and name, as shown in Listing 2. Node types can be formally defined via product-like codes in the spirit of [31], and can be used by the application to engage nodes in a more targeted way. The name of the node can be used to further differentiate between nodes of the same type, if desired. The type and name are static properties of the node (not specific to a service), and appear in the corresponding proxy object.

The concrete services to be provided by the node can be added and perhaps later removed, in a dynamic way, via the addition and subtraction operators, as shown in Listing 2. Service addition and removal triggers the invocation of the respective *activate*() and *passivate*() methods. Changes in the node's service provision are automatically reflected in the respective proxy object.

Listing 2: Implementing node-level behavior.

```
node = Node("Quadcopter","A")
...
brightness = BrightnessSvc()
...
node += brightness
...
node -= brightness
```

4.3 Coordinator object

Mission control is performed via an object of the singleton *Coordinator* class. This object supports different methods that can be used to inspect the mission group, form teams and control individual nodes but also entire teams, as needed. These aspects are discussed in more detail below.

4.4 Group inspection and node control

The mission group is accessible through the *group* object. This is part of the *Coordinator* object, and contains the proxies for all nodes that currently participate in the mission group. The application can traverse this collection using standard Python iteration structures, and inspect the services provided by each node via the corresponding proxy. Listing 3 gives an example where the application prints the type and name of nodes that are added to or removed from the mission group. Later on, the application takes pictures via the nodes that provide the photo service. The last piece of code shows how to do the same by invoking the service without checking that it is actually supported. If the node does not support the service, a runtime exception will be raised, which in this case is simply ignored by the application.

Listing 3: Browsing the mission group.

```
def GroupUpdate(grp,node,mode):
  print "Count:" + str(grp.size())
  if mode == "NODE_ADDED":
    print "added " + node.name
                   + ":" + node.type
  elif mode == "NODE_REMOVED":
    print "removed " + node.name
                     + ":" + node.type
...
co = Coordinator(...)
co.group.setUpdateNotifier(GroupUpdate)

for n in co.group:
  for svc in n.services:
    if isinstance(svc,PhotoSvc):
      picture = svc.shootPicture()

for n in co.group:
  try:
    n.PhotoSvc.shootPicture()
  except:
```

Individual nodes can be controlled by invoking the methods and observing the properties of their proxies. Moreover,

the application can block until a given node provides a service or a given service property reaches a specific value. This is done via the *waitOn*() method of the *Coordinator* object, which takes as a parameter a waiting expression and a timeout.

Listing 4: Node-level control.

```
n = ...
n.MobilitySvc.goto(pos)
co.waitOn(n,property=(MobilitySvc.dist<1.0),0)
print n.MobilitySvc.pos
pic = n.PhotoSvc.shootPicture()
detect_something(pic)
```

As an example, Listing 4 shows how to use the mobility service in order to instruct a node to move to a certain position, wait until it reaches that position (within a certain distance tolerance), and then take a picture via the photo service, which is then processed to detect an object or phenomenon of interest (note that detection could be performed locally, by the node itself, again through a suitable service). A timeout of 0 is used, meaning that the application intends to block until the waiting condition is satisfied. In case the node fails, *waitOn*() will return, and an exception will be raised when the application attempts to access the proxy.

4.5 Team formation and control

Different teams can be introduced via the *defTeam*() method of the *Coordinator* object. This takes as a parameter a membership expression, which may refer to node-level properties, service types and service-level properties, and returns a *Team* object. Teams are maintained behind the scenes so that they contain proxies for nodes that match the membership expression. One can browse the team and install a notifier for the corresponding membership updates, in a similar way this is done for the mission group. Listing 5 shows how to define a team for nodes that provide the photo service, with a notifier for printing the type and name of its members.

Listing 5: Team creation.

```
def PrintTeam(team):
  print "Count:" + str(team.size())
  for n in team
    print n.name + ":" + n.type

t = co.defTeam(node=(any),
               service="PhotoSvc",
               property=(any))
t.setUpdateNotifier(PrintTeam)
```

Services that are common among the members of the team are promoted to the team level, and the respective interfaces become directly accessible via the respective *Team* object. Team-level services can be inspected in the same way this is done for individual nodes (proxies). The invocation of a team-level method transparently leads to the invocation of the respective method at all members of the team. The results are returned as a dictionary of key-value pairs, where the key is the node object and the value is the result of the respective method call. Team-level properties are supported/accessed in the same way. Finally, as this is the case for individual nodes, it is possible to wait on team-level conditions.

Notably, service promotion allows the programmer to control an entire team in a straightforward way, (almost) as if it were a single node. In the spirit of Listing 4, Listing 6 shows how to move the photo team, which was created in Listing 5, at a certain location, print the positions of individual members, and take a team-level picture.

Listing 6: Team-level control.

```
t = ...
t.MobilitySvc.goto(pos);
co.waitOn(t,property=(MobilitySrv.dist<1.0),0)
print t.MobilitySvc.pos
pictures = t.PhotoSvc.shootPicture()

for node,pic in pictures.iteritems():
  detect_something(pic)
```

Instead of having all team members go to the same position as in the above example, team-level movement can be customized to maintain a specific formation. This can be achieved by assigning a suitable "offset" to each team member, which is used locally by the node to derive the actual target position relative to the team's position that is specified via the team-level *goto*() method. The offset can be calculated by the coordinator, depending on the desired formation, and can be adapted dynamically via the team-level update notifier when nodes join or leave the team. We do not show such an example for brevity, but it should be fairly obvious how this functionality can be implemented based on the above code examples. We have currently written code-templates for line, circle and grid formations, which can be reused in different applications. It is straightforward to add support for more complex formations.

Finally, it is possible to define teams in a fully controlled manner, by handpicking specific nodes based on their name. In this case, the team will be populated only when one or more of these nodes join the mission group. This mode of operation can be desirable if the application functionality relies on the services/capabilities of specific nodes, rather than on the formal type of a service independently of the specific node instance that provides it.

4.6 The TeCoLa environment

All the mechanisms of TeCoLa are implemented on top of the standard Python environment (without touching it). They perform behind the scenes all the work that is required to provide the above functionality to the programmer. Reflecting the coordinated flavor of the programming model, TeCoLa is internally structured in two environments, for the node and coordinator functions, respectively.

The node environment is used to instantiate and run the *Node* object. It joins the mission group, advertises the provided services, intercepts service method call requests, executes the service methods and returns the replies. It also propagates the updated values of dynamic service properties to the coordinator environment.

The coordinator environment is responsible for running the application logic, via the *Coordinator* object. It creates proxy objects for the nodes that join the mission, and populates the mission group and team objects accordingly. It also implements the method invocation and property update between proxies and the *Node* objects, by interacting with the corresponding node environments in a transparent way.

The communication between the coordinator and node environments is performed using an own transport protocol, which provides support for coordinated group management

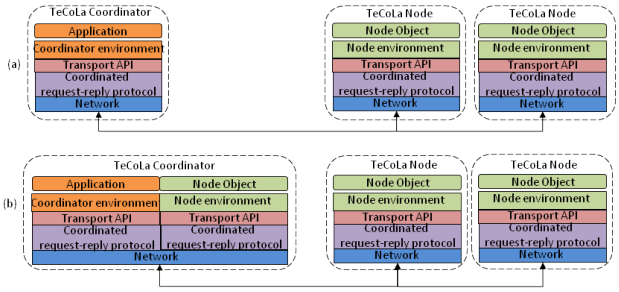

Figure 7: Supported TeCoLa configurations: (a) master-slave configuration, (b) peer-to-peer configuration.

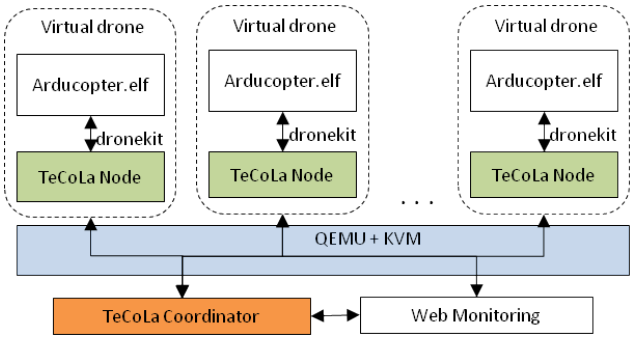

Figure 8: Architecture of the simulated drone environment.

and request/reply interactions efficiently on top of a wireless medium [24]. But note that our implementation is not bound to this particular network protocol, which could be replaced with any other transport with similar functionality.

The protocol exploits the broadcast-nature of the wireless medium, and performs 1-to-N request/reply exchanges with the minimum number of message transmissions and at low latency, while avoiding network contention among the communicating parties. The coordinator environment maps every node- or team-level method invocation to a corresponding request/reply interaction, addressing the nodes that need to be involved in each case. The updates of service property values are collected using a polling scheme, via periodic queries performed by the coordinator environment in the background without any explicit request from the application—these queries are also used to confirm the liveness of the nodes that are part of the mission group, and to update the group in case of node failures. The *Coordinator* object has special methods for setting the desired polling rate, separately for each node, team or service, if so desired.

In addition, the protocol allows new nodes to join the group dynamically. At the level of TeCoLa, the application can invoke this process explicitly or instruct the runtime to do this periodically, via corresponding methods of the *Coordinator* object. When the coordinator environment is notified about the addition of a new node, it creates a proxy and adds it to the mission group and possibly to one or more teams. The transport protocol can also detect node failures. These are reported to the coordinator environment, leading to the removal of the corresponding proxies from the mission group and teams.

TeCoLa can be used in two different system configurations, illustrated in Figure 7. In the so-called "master-slave" configuration a distinguished node only features the coordinator environment and all other nodes feature the node environment, whereas in the "peer-to-peer" configuration the node with the coordinator environment also features the node environment (provides some services on its own). In both cases, if the node with the coordinator environment fails, other nodes enter a "fail-safe" state. We are currently extending the "peer-to-peer" configuration so that, in case of a coordinator failure, another node can take over as a new coordinator. The underlying transport protocol already provides this functionality. We are now in the process of introducing suitable checkpointing hooks at the level of TeCoLa in order to replicate the coordinator environment and communicate updates to the replicas, in the spirit of a primary-backup scheme [12].

5. FUNCTIONAL TESTING

We have performed extensive functional tests of TeCoLa using a simulated setup that includes multiple communicating virtual drones, as shown in Figure 8. Each virtual drone consists of a Linux virtual machine (VM) that implements a realistic software-in-the-loop simulation of an Arducopter equipped with the APM autopilot [3]. The autopilot is connected to an Arducopter flight dynamic model, responsible for the flight dynamics simulation. The node environment runs on the VM of the virtual drone, and communicates with the autopilot via the MavLink protocol [4] through the Drone-kit API—in the same way as if it where running on a companion board on a real Arducopter. The different virtual drones run on top of the QEMU emulator on a KVM-enabled Linux server, and networking between them is achieved through the standard KVM network bridge. In this setup, we employ the master-slave configuration of TeCoLa, with the coordinator environment running natively on the local machine. For visualization purposes, the current positions of the virtual drones are communicated (via WebSockets) to a web-based application, which draws corresponding markers on a map, using the standard google maps API [5]. Notably, the entire TeCoLa environment (and application code) that runs on top of these virtual drones, could be transferred to real drones with very few modifications.

Next, we briefly describe how TeCoLa was used to implement a forest fire detection and response scenario, in the spirit of the one in Section 2. We consider three different types of drones: the scanner, the observer and the fire extinguisher. Each type comes with different flight capabilities and equipment, and provides different special services. The scanner can fly long distances at high altitude, carries high-resolution infrared and visible light cameras, and provides a fire detection service. The observer is a cheap lightweight drone with long hovering capability, equipped with a visible light camera, and provides a fire tracking service. Finally, the extinguisher is a heavyweight drone that can cover relatively short distances, is equipped with a water cannon and a tank with firefighting chemicals, and provides a fire extinguisher service.

A predefined forest area is continuously monitored by scanners from high altitude. At a certain location in the forest, there is a pavilion of the fire department where a number of inactive observers and extinguishers are stationed. When a fire outbreak is detected by the scanners, a number of observers are sent out to delineate the fire front and to produce more detailed information that can be used to guide

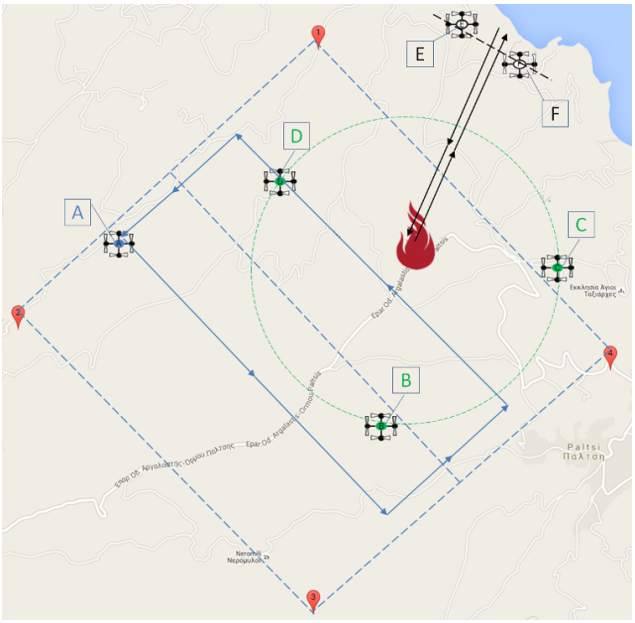

Figure 9: Visual snapshot of the fire detection and response scenario, using a configuration with one scanner, three observers and two extinguisher drones.

firefighting operations. At the same time, the scanners keep scanning the entire forest area, as before, to detect new fire spots (simultaneous fire outbreaks are quite common in case of arson attempts). Based on the information produced by the observers, the nearby fire extinguishers are subsequently sent out to release chemicals and/or water on top of the fire, and then return back to the station in order to reload.

To support this scenario, the application employs three teams, for the fire detection, fire tracking and fire extinguisher service, respectively. Each team maintains a different formation suitable for its purpose. More specifically, the scanner team performs a high-altitude flight, scanning the area of interest in consecutive passes, from one border to the other, until the entire area is covered; note that the number of passes that need to be is depends on the number of team members. The observer team follows a circular formation around the epicenter of the fire, with the radius being adjusted as the fire grows or spreads in different directions. Finally, the team of the fire extinguishers maintains a parallel line formation, and moves back and forth between the station area (to re-load) and the location where the fire is most intense (to drop chemicals/water).

A snapshot of an indicative scenario is shown in Figure 9 for the mountain area of Pelion in Greece, where large parts of the forest are practically inaccessible from the ground. On top of the map, we superimpose the paths and formation pattern of the different teams. In this particular case, only one scanner is used (drone A), which surveils the area in two passes, each covering a separate land stripe. A fire has already been detected, which is tracked by three observers (drones B,C and D) hovering at different points on the periphery of a circle. Finally, two extinguishers are employed to fight the fire (drones E and F), shown on their way from the station/reload area to the fire area, on the left and right hand side of the path that is specified by the application.

This non-trivial functionality can be implemented in a clean and straightforward way using the TeCoLa primitives: teams, team update notifiers, team-level service methods and team-level service properties. The application logic is about 100 lines of code. Note that the same code will work, without any modification, irrespectively of the number of drones in each team.

6. RELATED WORK

Recently, several programming abstractions and languages have been proposed to simplify the development of applications using mobile robots. In the following, we briefly review indicative work, and compare it with our approach.

The Robotic Operating System (ROS) [29] adopts a modular and loosely-coupled architecture for organizing the different functions of a robotic system. The basic functional components are referred to as nodes, and can implement a variety of operations, from the handling of hardware resources to path-planning and navigation algorithms. Nodes interact with each other by exchanging messages using a publish/subscribe mechanism. A node may also provide services that can be invoked in an explicit way, as in TeCoLa. The Mission Oriented Operating Suite (MOSS) [27] follows a similar approach to ROS. In this case, the individual functional components are implemented as separate processes, which communicate with each other indirectly, through a server that routes the messages being published to the respective subscribers. ROS and MOOS are mainly designed for single robots, and focus on decoupling the different components of a robotic platform. Still, given their pub/sub architecture, they could be adapted to support distributed multi-robot systems where communication among nodes/processes takes place over a network. TeCoLa operates at a higher level, and could be implemented on top of such distributed pub/sub frameworks.

In URBI [9], a robot comprises a set of abstract device objects. Each device is associated with a sensor or actuator, and implements a set of services for accessing the particular resource. URBI follows a client/server architecture whereby the robot act as a server, and the client connects to it via telnet. The client then uploads and runs the application program, which is written as a script that accesses one or more devices of the robot, as needed. While the resource abstraction is similar to that of TeCoLa, resource access is a strictly local operation and does not occur over the network. Another major difference is that URBI only targets single robots, without providing any support for coordinated group/team operations.

Other frameworks such as [14] and [15] also follow a service-oriented approach. In this case, robotic resources and capabilities are captured in the form of web services [6], which can be invoked in a structured and transparent way from a remote machine. However, unlike TeCoLa, these frameworks are designed to operate with a fixed and well-defined group of robots and resources, and do not offer any advanced programming support for service/team dynamics.

The Miro [18] middleware employs an object-oriented architecture, structured in three layers. The first layer provides platform-specific service abstractions for the sensors and the actuators of the robot, the second layer exports these services via the CORBA interface definition language, and the top layer provides higher-level services such as navigation and path planning. The programmer accesses these ser-

vices via the standard CORBA object protocols. Miro has similarities with TeCoLa in terms of the resource abstraction model, but uses a more complex and language-neutral middleware for service invocation. Also, Miro targets static robotic groups without allowing the dynamic addition of resources at runtime, and the burden of coordinating a robotic team is left entirely to the programmer.

Karma [17] is an application programming model targeting resource-limited micro-aerial vehicles that do not have any communication capability when operating in field. A pool of drones is stationed in the so-called hive, which has a central data store for keeping the information generated by the drones. The application logic consists of drone behaviors along with activation predicates and progress functions. The activation predicates are evaluated based on the information that is currently available in the data store, and in turn can lead to the activation of a behavior. The hive allocates one or more drones to execute the activated behavior, which then fly out to perform the assigned mission. When the drones return, they offload the data collected/produced to the data store, and the respective progress function is evaluated to assess application progress. Like TeCoLa, Karma adopts a coordinated approach, but the coordination is rather coarse and cannot be adjusted in a flexible way—once a drone leaves the hive, it operates independently from other drones, and does not communicate with the data store during the course of a mission.

In Meld [7], an entire swarm of robots is programmed as a single entity, using a declarative language. The resources provided by each robot as well as the desired high-level achievements of the swarm are represented through facts and a set of rules. The rules are evaluated progressively by the swarm to produce new facts, until the top-level achievement is satisfied. Rules are propagated to the entire swarm, whereas facts are distributed among the robots based on their capabilities. If a computation requires facts that are not locally available, it retrieves them from other robots in the neighborhood. Thanks to the declarative nature of Meld, the programmer can formulate relatively simple missions in an easy way. However, unlike in TeCoLa, one cannot control/steer robots in an explicit way, in order to customize the swarm behavior according to application-specific needs or to support more dynamic missions.

The Proto language [8] has its roots in the parallel computing domain. The space where the robots operate is abstracted as a so-called amorphous medium. Each point of the medium is a computational device that is associated with a tuple of values, which are internally mapped to one or more mobile robots. Application-level operations are performed on the medium, while a functional oriented compositional approach is used to transform space operations into instructions for the individual devices, and ultimately into commands for each mobile robot. Proto comes with support for multi-robot operation and coordination, but this is implicit and at the granularity of spatial neighborhoods. In contrast, TeCoLa supports more versatile team formation, based on services and (dynamic) properties, and can more purposefully exploit heterogeneity.

Voltron [26] targets active sensing applications where system behavior is adapted according to the sensor data. Its key abstraction is that of a single virtual drone, which features the entire set of programming primitives. The application is written for this virtual drone, specifying a number of sensing operations that need to be performed at certain locations. The underlying system coordinates the individual drones that are available in order to perform these sensing operations, based on a virtual synchrony mechanism [20]. Voltron takes care of dynamic group management, and the programmer does not need to be aware of the number of available robots. However, the details of swarm scheduling and formation cannot be customized by the application. There is also no support for resource heterogeneity.

Buzz [28] is a language for programming heterogeneous swarms of robots. Similar to TeCoLa, the resources of each robot are accessed via method calls. In addition, it is possible to create swarms based on the resources found in the neighborhood of a robot, and the application can invoke both swarm-level and robot-level operations. One major difference is that Buzz follows a decentralized approach where swarm formation and control is performed in a peer-to-peer manner, whereas in TeCoLa this process is driven by the coordinator. Also, while Buzz allows for a robot to share selected state information with the swarm, this must be done in an explicit way using special broadcast-like operations (virtual stigmetry). Moreover, the entire application logic must be pre-installed on every robot, which makes it harder to employ robots on-demand, or to use the same robot for different or very dynamic application missions without reporgramming it.

TeCoLa combines some features of Voltron and Buzz, along with architectural elements of Miro and URBI in order to achieve an easy and versatile coordination of robotic groups where the resources may change dynamically based on the needs of dynamic missions.

7. CONCLUSIONS AND OUTLOOK

Writing applications that can exploit several and different types of mobile robots is a daunting task. TeCoLa simplifies development by offering suitable primitives, which explicitly target heterogeneity and enable control at the team level. The complexity of team formation and team dynamics, due to the addition and removal of individual roots at runtime, remains hidden behind the scenes, while providing suitable hooks to the programmer for adapting the application behavior as needed. We have implemented a working version of TeCoLa for a Python environment, and have tested its functionality using a software-in-the-loop simulation environment, through a non-trivial application scenario, which can be supported at a modest programming effort.

We are currently working on an extended implementation that can tolerate coordinator failures in a transparent way, allowing the application to continue its execution without a major interruption or reset. We also plan to support long-range communication between the coordinator and the command and control center. In addition, we wish to enrich the syntax of team membership and wait expressions in order to allow for more advanced declarative functionality, as well as to support mobility formation in a more "institutional" manner, as a first-class aspect of TeCoLa, rather than through application-level code templates. We also wish to explore the integration of TeCoLa with stationary sensor/actuator infrastructures, so that these can be seamlessly exploited by the application. Last but not least, we are planning to test TeCoLa with real UAVs, in a suitably controlled/contained physical testbed.

8. ACKNOWLEDGMENTS

This work has been partially supported by the Horizon 2020 Research and Innovation programme of the European Union, under project RAWFIE, grant agreement No 645220.

9. REFERENCES

[1] http://www.parrot.com/usa/.
[2] http://myziphius.com/.
[3] http://ardupilot.org/ardupilot/index.html.
[4] http://qgroundcontrol.org/mavlink/start.
[5] https://developers.google.com/maps/.
[6] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer Science Business Media, 2004.
[7] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *International Conference on Intelligent Robots and Systems*, 2007.
[8] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous-space programs for robotic swarms. *Neural Comput & Applic*, 19(6):825–847, 2010.
[9] J.-C. Baillie. URBI: towards a universal robotic low-level programming language. In *International Conference on Intelligent Robots and Systems*, 2005.
[10] M. Bell. Service-oriented modeling. *John Willey & Sons*, 2008.
[11] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.
[12] N. Budhiraja and K. Marzullo. Tradeoffs in implementing primary-backup protocols. In *7th Symposium on Parallel and Distributed Processing*, 1995.
[13] D. W. Casbeer, D. B. Kingston, R. W. Beard, and T. W. McLain. Cooperative forest fire surveillance using a team of small unmanned air vehicles. *International Journal of Systems Science*, 37(6):351–360, 2006.
[14] Y. Chen and X. Bai. On robotics applications in service-oriented architecture. In *28th International Conference on Distributed Computing Systems Workshops*, 2008.
[15] Y. Chen, Z. Du, and M. García-Acosta. Robot as a service in cloud computing. In *5th International Symposium on Service Oriented System Engineering*, 2010.
[16] E. Christopoulou and A. Kameas. Ontology-driven composition of service-oriented ubiquitous computing applications. In *3rd International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*, 2004.
[17] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh. Programming micro-aerial vehicle swarms with karma. In *9th Conference on Embedded Networked Sensor Systems*, 2011.
[18] S. Enderle, H. Utz, S. Sablatnög, S. Simon, G. Kraetzschmar, and G. Palm. Miro: Middleware for autonomous mobile robots. In *In Telematics Applications in Automation and Robotics*, 2001.
[19] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.
[20] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. Virtual synchrony guarantees for cyber-physical systems. In *32nd International Symposium on Reliable Distributed Systems*, 2013.
[21] M. Forte, W. L. de Souza, and A. F. do Prado. Using ontologies and web services for content adaptation in ubiquitous computing. *Journal of Systems and Software*, 81(3):368–381, 2008.
[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Object-Oriented Programming*. Springer, 1993.
[23] S. Griffiths, J. Saunders, A. Curtis, B. Barber, T. McLain, and R. Beard. Maximizing miniature aerial vehicles. *IEEE Robotics Automation Magazine*, 13(3):34–43, 2006.
[24] M. Koutsoubelias and S. Lalis. Coordinated Broadcast-based Request-Reply and Group Management for Tightly-Coupled Wireless Systems. In *22nd International Conference on Parallel and Distributed Systems*, 2016.
[25] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Conference on Programming Language Design and Implementation*, 1988.
[26] L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi. Team-level programming of drone sensor networks. In *12th Conference on Embedded Network Sensor Systems*, 2014.
[27] P. M. Newman. Moos-mission orientated operating suite. *Technical Report, Ocean Engineering Dept., Massachusetts Institute of Technology*, 2002.
[28] C. Pinciroli, A. Lee-Brown, and G. Beltrame. Buzz: An Extensible Programming Language for Self-Organizing Heterogeneous Robot Swarms. *arXiv:1507.05946*, 2015.
[29] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.
[30] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, 2010.
[31] D. J. Russomanno, C. Kothari, and O. Thomas. Sensor ontologies: from shallow to deep models. In *37th Southeastern Symposium on System Theory*, 2005.
[32] V. Šmídl and R. Hofman. Tracking of atmospheric release of pollution using unmanned aerial vehicles. *Atmospheric Environment*, 67:425–436, 2013.
[33] C. Zhang and J. M. Kovacs. The application of small unmanned aerial systems for precision agriculture: a review. *Precision Agric*, 13(6):693–712, 2012.