
Paul Glotfelter, Travis Eichelberger, and Patrick J. Martin

Abstract—This paper presents a cloud-computing inspired framework that facilitates the programming of a deployed cyber-physical system. This framework, PhysiCloud, uses a novel combination of abstractions that hide the implementation details of the underlying cyber-physical system. Additionally, the framework is designed to operate on low-power, mobile systems with resiliency to network failures. Using this system, a controls application developer can focus on their algorithm development and its information dependencies, rather than issues of low level scheduling and communication.

I. INTRODUCTION

Inexpensive, embedded computing systems, environmental sensors, and actuators are used frequently in industrial and consumer products. From home automation to unmanned transportation, cyber-physical systems (CPS) are set to revolutionize how humans interact with the virtual and physical worlds [1]. Control engineers will need to develop safe, reliable, and user friendly CPS in diverse application domains, such as building control systems and unmanned vehicles, e.g. [2], [3]. However, developing and deploying control applications for these systems is complicated by heterogeneous computing, networking, sensing, and actuating subsystems. There is a need to develop software frameworks that unify these complex systems such that they can be “programmed” much like a single multi-core processor is programmed in Java, C++, or other standard language.

To approach this issue, one can draw inspiration from cloud computing [4], [5], where complex networks of high-end servers are pooled together using new software and hardware architectures. The cloud computing abstraction allows programmers and users to interface to these high-speed, networked systems as if they were local computational resources. This computing paradigm has three service delivery models: software-as-a-service (SaaS), platform-as-a-service (PaaS), and infrastructure-as-a-service (IaaS). To facilitate these service models, cloud computing middleware manages the underlying networks and schedules distributed applications across the servers. Additionally, these software systems utilize hypervisors [6] to allocate and manage virtual machines that accommodate different application types within the cloud.

Many current and future CPS applications involve mobile systems that have finite energy and must operate in dynamic environments. The mobile agents within these CPS need to run low-power, memory-constrained ARM embedded systems to maximize runtime of the deployed system. However, current cloud computing tools, such as the open-source Xen Hypervisor, have limited support for ARM based systems. Additionally, the limited memory and storage on these mobile systems makes the launching of a full virtual machine difficult, if not infeasible.

The contribution of this paper is a new software framework, PhysiCloud, that creates a PaaS across a CPS composed of low-power, mobile systems. PhysiCloud provides an input-output programming model that is natural for developing control algorithms with networked information requirements. Additionally, this framework creates an execution runtime for CPS control applications that is resilient to networking failures that are common in mobile systems.

There is increasing interest in utilizing cloud- and pervasive- computing technology for CPS applications. The work in [8], [9] proposed the notion of information-acquisition-as-a-service cloud computing. To enable this service, the authors proposed the “virtual vehicle” abstraction. A virtual vehicle represents an application that must run in the cloud and it is monitored by a virtual vehicle monitor. This approach to managing a CPS requires the use of a custom Xen hypervisor, which limits its availability on ARM systems. Additionally, it considers the mobile devices as information harvesters rather than elements that can provide computing services for a higher-level application. PhysiCloud considers these types of systems important for carrying out computations, especially if they are located near the physical plant being controlled. Furthermore, our framework is able to run on memory-constrained ARM systems where the creation and management of a full virtual machine may consume considerable resources.

The authors of [10] proposed a pervasive computing framework for managing a collection of heterogeneous sensors and computers. This system was able use different communication media, i.e. WiFi, Zigbee, and Ethernet, and execute across a diverse network of systems. The authors created a Java-based programming abstraction that allows application developers to utilize the computing and sensing resources within the system. PhysiCloud differs from this work by adopting an input-output programming model that works naturally with developing CPS controller code.

This paper is organized in the following way: Section II presents the core system design. The implementation details of this design are given in Section III. Section IV evaluates PhysiCloud’s resiliency and demonstrates PhysiCloud’s exe-
cution of a distributed control application. Finally, we discuss
the future directions for PhysiCloud in Section V.

II. SYSTEM DESIGN

PhysiCloud aims to bring a PaaS resource and program-
merng model for developing CPS control applications. This
framework provides the necessary middleware to enable a
future high level specification language for developers. This
approach is similar to the Google App Engine\(^1\), which allows
anyone to develop applications that target Google’s cloud
system. PhysiCloud will give CPS researchers and developers
the ability to deploy “mini-clouds” that can manage their
heterogenous, low-power systems as if they were a local
resource.

The design of PhysiCloud has the following goals:

• Provide a simple programming abstraction for control
application development
• Execute on systems with limited computational and
power resources
• Create a resilient runtime environment that adapts to
structural changes.

To achieve these goals, PhysiCloud’s the design draws
inspiration from cloud computing and models of mobile,
concurrent computation.

A. Core System Elements

PhysiCloud is composed of a collection of cyber-physical
units, or CPUs, that each have computing, sensing, and,
optionally, actuation capabilities. Figure 1 illustrates a par-
ticular instance of a PhysiCloud enabled CPS. The use of
the CPU acronym is intentional, as we want to abstract the
underlying elements as devices on which control tasks are
allocated for execution on the cloud, much like processes
are allocated on multi-core processors. We consider the
“busses” among the CPUs as the communication medium,
such as 802.11 WiFi or Ethernet. Each CPU maintains its
local resources; however, the core of PhysiCloud provides
knowledge about the other available cyber-physical resources
in the cloud.

Each processor, sensor, and actuator is a resource that
may be used by a CPS control application. PhysiCloud
manages the interconnect among these CPUs, which allows
a control application designer to focus on the input-output
relationships among the various control modules required for
execution.

To model the cyber-physical resources and code execution,
we leverage the asynchronous distributed pi-calculus (ADPi)
[11]. This model is related to the concept of software
actors [12], [13]; however, the syntax of the model in-
cludes networking and location operations. ADPi considers
all resources in a distributed computing environment
as information channels. Processes that execute within a
distributed environment communicate via these channels and
may operate independently. The ADPi resource model allows
for flexible reorganization of PhysiCloud when the makeup

\(^{1}\)http://cloud.google.com/AppEngine

Fig. 1. An illustration of the core elements of a PhysiCloud instance. In
this case, the cloud is made up of a mobile robot, a sensor node, and a
standard laptop. This diagram shows the resources available within each
PhysiCloud enabled CPU.

of the system changes, such as when a CPU loses power.
In PhysiCloud, a CPS control application is a collection of
tasks that each subscribe and publish to resource channels.
Tasks are threads of execution that accept information from
a finite number of input channels and produce results on a
single channel.

To illustrate the model of a task, consider the diagram
of a Kalman filter, shown in Figure 2. This Kalman filter
task depends on the IMU and GPS resources, which usually
are supplied via local software modules. However, if they
are supplied by an external element in the deployed CPS,
PhysiCloud routes the information across the network to this
task.

Assume that we want this Kalman filter task to execute
every 200 ms using the most recent information from the
IMU and GPS resources. Using PhysiCloud’s syntax, we
can easily load this task into the system runtime with the
following code:

\begin{verbatim}
(task :name 'kalman-filter
 :update-time 200
 :function (fn [imu gps]
 <Compute filter...>)
 :produces 'state-estimate')
\end{verbatim}

This code tells PhysiCloud the task’s name, the update
period, the function that should execute on the data, and

Fig. 2. This figure illustrates the task abstraction in PhysiCloud. This
Kalman Filter task requires information from the IMU and GPS resources
and produces a state estimate resource as its output, \([\hat{x}, \hat{y}, \hat{\theta}]\).

Assume that we want this Kalman filter task to execute
every 200 ms using the most recent information from the
IMU and GPS resources. Using PhysiCloud’s syntax, we
can easily load this task into the system runtime with the
following code:

\begin{verbatim}
(task :name 'kalman-filter
 :update-time 200
 :function (fn [imu gps]
 <Compute filter...>)
 :produces 'state-estimate')
\end{verbatim}

This code tells PhysiCloud the task’s name, the update
period, the function that should execute on the data, and
what the task produces. The system automatically connects
the imu and gps dependencies to the task and creates the
new resource, state-estimate. This new resource is
available for consumption by any other task in PhysiCloud,
be it local or on another CPU. For example, the estimate of
the robot state should be utilized by a control algorithm, but
it may also be useful to a user interface located on another
system. These example consuming tasks could be written:

(task :name 'user-interface'
  :update-time 500
  :function (fn [state-estimate]
    <Display robot...>))

(task :name 'go-to-goal'
  :type event
  :function (fn [state-estimate]
    <Compute control law...>
    :produces 'control-vector')

By abstracting resources as information channels, tasks can
execute anywhere within a PhysiCloud instance.

Additionally, the prior sample code demonstrates the two
types of execution modes for tasks: 1) time-driven and 2)
event-driven. Time-driven tasks execute periodically, based
on their local CPU’s operating system clock. Event-driven
tasks execute only when information from one of their depen-
dencies arrives as an input. The go-to-goal task above
will compute the control law only when new information
arrives from its state-estimate channel. However, the
user-interface task will always execute its function
every 500 ms using the most recently buffered value for the
state estimate of the robot.

B. Networking Model

To extend the notion of resource channels across a CPS
network, PhysiCloud adopts an overlay networking model
[4]. This approach lets users specify access to networked
resources as if they were locally enumerated on the particular
CPU. All networked resources are managed by a special
server within PhysiCloud, called the monitor. The monitor
keeps track of all subscriptions to required networked re-
sources and supplies the required information to resource
subscribers.

Consider, again, the instance of a PhysiCloud system in
Figure 1 where three CPU elements each have a set of
available resources. If one wants to deploy the PhysiCloud
application shown in Figure 3, the state estimate resource
will need to be shared across the network via an overlay
channel.

In this example, assume that the kalman-filter task
is assigned for execution on the robot CPU. Additionally,
the user-interface task is allocated to the laptop
CPU and it needs the state estimate information from the
kalman-filter task. While this information is supplied
to the user interface, the robot will need to execute the
go-to-goal behavior with this same information. Physi-
Cloud automatically generates the communication facilities
needed to connect the state-estimate resource over the
network for each consuming task, as shown in Figure 4.

In this illustration, the state estimate resource is a channel
located within the robot CPU; however, PhysiCloud con-
ects it logically to the laptop CPU. During execution any
messages within the state estimate channel are immediately
delivered to the go-to-goal task and are also forwarded
to the user-interface task that subscribes to it. By uti-
лизing the concept of a network overlay, the logical structure
of a CPS application can treat any resource as if it were
locally available.

III. SYSTEM IMPLEMENTATION

To enable the key features in PhysiCloud, every CPU must
manage internal threading and external networking. This
section discusses the internal execution mechanisms as well
as the network protocols required for operation. Additionally,
we discuss the practical deployment of the current version
of the software.

A. System Kernel

Each PhysiCloud CPU maintains its internal and external
interactions with a system kernel. This kernel is responsible
for task scheduling and the allocation (or, de-allocation) of
internal memory channels. Additionally, each CPU kernel
provides software instrumentation to collect data about the
deployed system, such as architecture types, average task
execution times, or network latency. Most importantly, the
kernel manages the network infrastructure to support net-
worked resource subscriptions.

Tasks are implemented as light-weight processes that
subscribe to resource channels required for execution. Time-
driven tasks are assigned to an internal scheduler, whereas

---

**Fig. 3.** A PhysiCloud application that displays a robot’s location on a
graphical interface while the robot executes its Go-To-Goal behavior.

**Fig. 4.** An illustration of how networked resources are logically connected
across PhysiCloud CPU elements.
event-driven tasks attach to resources with a callback function that responds when new data arrives. If a networked resource is required, the kernel handles the resource discovery and the allocation of support communication to deliver the data.

B. Network Structure

Since control applications require reliable communication of data, we use a TCP/IP server-client architecture to ensure message delivery to consuming tasks. Each PhysiCloud instance is composed of a single server, called the monitor, and a collection of clients. The monitor is a unique, dynamic server that manages all networked channels in the deployed system. Initiating PhysiCloud requires a two phase process: 1) network discovery and 2) selection of a monitor. Upon startup, each CPU transmits their IP address via UDP broadcasts across a pre-determined subnet. Each member that receives the broadcast message builds up a table of all neighbors in the network.

Once every CPU is mutually aware of all other members of the network, they asynchronously execute an algorithm to determine which CPU is capable of hosting the monitor. The elected CPU then initializes the monitor and the remaining CPUs connect to this newly established monitor. In the current implementation, this allocation algorithm chooses a CPU at random to serve as the monitor. In the future, we plan to implement more intelligent allocation algorithms that would place an emphasis on power conservation or CPU mobility.

After the monitor is established, it creates a single “kernel channel” to which all clients subscribe and transmit information about their available resources. At this point, the entire system is aware of its cyber-physical resources and can accept control applications. Whenever an application is deployed, its tasks are assigned to the available CPUs for execution.

C. Network Management and Resiliency

Once PhysiCloud’s core system is built across the network, several network operations occur during the execution of a control application. One such operation is networked resource allocation and management. If a task needs a networked resource, its local CPU checks the availability of the resource. If any are found within PhysiCloud, the task’s CPU selects the resource with the lowest network latency. The kernel constructs a logical channel for the supplier CPU to publish new data. This data arrives at the dependent task’s CPU and is routed internally to local subscribers.

Figure 5 shows the relationship among the monitor and the two client CPUs in our example. The “state-estimate” provided by the robot CPU is generated locally, but its information is published via TCP/IP to the monitor. After the monitor receives this message, it sends it to all subscribers of the “state-estimate” channel. Once the client CPU receives a state estimate message over this connection, it internally routes the message through a local representation of the channel to the User Interface task. Note that at the same time, the robot CPU is routing this message internally to the Go-To-Goal task.

This implementation for PhysiCloud’s networking model has a balance of tradeoffs: it provides system flexibility and reliable delivery, but increases latency. Alternatively, the entire system could use a custom UDP protocol that enables direct messaging among the tasks and their supporting kernels. However, since UDP is connectionless, it would require more processing overhead to ensure that messages are processed in order at the receiving end. Additionally, since our design uses asynchronous message passing, which is similar to email communication, TCP/IP is better suited for the network management.

Since the monitor is usually mobile and has finite energy, its failure due to power loss or destruction is likely. Consequently, PhysiCloud is implemented so that it can recover from the inevitable loss of this monitor. Consider again the scenario from Figure 5. If the sensor node CPU loses power, the routing of the state estimate will fail. However, the client CPUs detect if a monitor is live through heartbeat messages. Once a monitor failure is observed, the following events occur to repair the PhysiCloud network:

- The tasks are put into a warning state that notifies the user of possible network dependency losses.
- All currently running tasks with a network dependency may be halted while the overlay network is regenerated.
- The protocol for establishing a monitor is re-executed.
- A new monitor is established and all halted processes are allowed to resume communication.

During the recovery process, executing tasks are optionally halted, since the operation is left up to the application developer. In some control tasks, such as ones that are designed to handle data delays or faults, the developer may decide to let the task continue execution while the network is being repaired.

D. System Deployment

PhysiCloud is implemented using a functional programming language, Clojure2, which executes on the Java Virtual...
Machine (JVM). Since PhysiCloud targets the JVM, it is deployable on many different types of architectures: from powerful servers to low-power ARM systems. These ARM systems are a critical member of the PhysiCloud ecosystem, since they are commonly used in mobile robotics and as intelligent sensor nodes. Currently, PhysiCloud can operate on the following embedded Linux platforms: Udoo, BeagleBone Black, and Raspberry Pi. All code and documentation for this project is available at the PhysiCloud repository\(^3\).

IV. EVALUATION

To the best of our knowledge, there is no “state-of-the-art” benchmark for CPS cloud computing environments. The work of [8] requires the use of a hypervisor that cannot execute on our target systems; consequently, we are unable to deploy a similar system for testing. Additionally, the source code for [10] is unavailable, which prevents us from directly comparing it to PhysiCloud. Consequently, we evaluate PhysiCloud’s features using two feature tests: 1) a forced monitor failure test and 2) a deployed decentralized control algorithm.

A. Monitor Failure Test

PhysiCloud’s design expects that mobile systems will leave the operating environment due to power losses or physical removal. Client devices are able to come and go in the network without causing any issues with the management of PhysiCloud. However, if a monitor device fails, all networked resource functionality stops. Therefore, it is imperative that the monitor is re-established within PhysiCloud using the protocol described in Section III-C.

To test this monitor repair protocol, we loaded PhysiCloud onto three devices that communicated via ad-hoc 802.11b wireless interfaces: 1) two laptop computers and 2) a Udoo embedded ARM computer. Each device ran a collection of tasks that required information from the other members of the network. Once the three systems established the PhysiCloud kernel, we shut down a monitor system by stopping its local network. Once the three systems established the PhysiCloud network within a reasonable time and with a minimal loss of CPUs. In the future, we will add additional TCP error handling mechanisms to ensure that systems will reliably reconnect.

B. Decentralized Algorithm Test

To illustrate PhysiCloud’s ability to support control application testing on a CPS, we implemented the distributed optimization algorithm presented in [14]. This algorithm uses a hybrid distributed and centralized approach to achieve coordination among a team of mobile agents. The interesting aspect to this approach is that agents have full knowledge of their local state, but their neighbor data is produced by an aggregating agent somewhere else in the cloud. The agents compute their controller commands using this delayed neighbor state information. Since the agents must act autonomously and asynchronously, its implementation takes advantage of PhysiCloud’s networking and resource management.

In general, let each agent have a state \( x_i \in \mathbb{R}^n \), where \( i \in \{1, \cdots , N\} \). The agents in this algorithm minimize their local objective function subject to \( m \) global constraint functions:

\[
g(x) = \begin{pmatrix} g_1(x) \\ g_2(x) \\ \vdots \\ g_m(x) \end{pmatrix}
\]

where \( x \) is the vector of all agent states.

Our implementation of this algorithm uses a scenario where five mobile agents must cover a pentagonal area. Each agent is assumed to have a state \( x_i \in \mathbb{R}^2 \) and a quartic objective function

\[
f_i(x_i) = (x_i - \bar{x}_i)^4
\]

such that

\[
\begin{align*}
\bar{x}_1 &= \begin{bmatrix} 0.0 & 0.3 \end{bmatrix}^T \\
\bar{x}_2 &= \begin{bmatrix} 0.2 & 0.1 \end{bmatrix}^T \\
\bar{x}_3 &= \begin{bmatrix} 0.2 & -0.1 \end{bmatrix}^T \\
\bar{x}_4 &= \begin{bmatrix} -0.2 & -0.1 \end{bmatrix}^T \\
\bar{x}_5 &= \begin{bmatrix} -0.2 & 0.1 \end{bmatrix}^T
\end{align*}
\]

Additionally, the agents are subject to the following global constraints

\[
\begin{align*}
(x_{5,2} - x_{4,2})^4 - 0.3 & \leq 0 \\
(x_{5,1} - x_{4,1})^4 - 1.0 & \leq 0 \\
(x_{2,2} - x_{3,2})^4 - 0.4 & \leq 0 \\
(x_{1,1} - x_{2,1})^4 - 0.6 & \leq 0
\end{align*}
\]

These choices for local objective and global constraints should result in the agents approaching the points of a pentagon.

The aggregating agent collects state information from all agents and periodically sends updated state information for the agents to incorporate into their local controllers. This task subscribes to all of the agent state resources (denoted \( x_1, x_2, \text{etc.} \)) that the individual agents produce, as shown in the code snippet below:

\[^3\text{http://github.com/hypower-org/physicloud}\]
Currently, PhysiCloud provides a programming interface for control application development across a deployed CPS. To use the system, a developer needs to code their control algorithms in the implementation language, Clojure. However, our future work will develop a high level specification language and toolkit that abstracts these language details.

Once a high-level language is developed, there will be a need to “compile” this code to run on a deployed PhysiCloud instance. Our future work will explore this phase of CPS system programming by matching the desires of a system developer with the actual capabilities of the deployed CPS.

ACKNOWLEDGMENT

The authors thank the US National Science Foundation for their generous support of this project. Additionally, the authors thank Sam Nelson for further development and refinement of PhysiCloud, as well as Matt Hale and Magnus Egerstedt for helpful discussions about their algorithm implementation.

REFERENCES