Cyber-Physical Cloud Computing: The Binding and Migration Problem

Kirsch, C.*, Pereira, E.†, Sengupta, R.†, Chen, H.‡, Hansen, R.†, Huan, J.†, Landolt, F.*, Lippautz, M.*, Rottmann, A.*, Swick, R.†, Trummer, R.* and Vizzini, D.†
*Department of Computer Sciences, University of Salzburg
Email: {christoph.kirsch,florian.landolt,michael.lippautz,andreas.rottmann,rainer.trummer}@cs.uni-salzburg.at
†Department of Civil and Environmental Engineering, University of California, Berkeley
Email: sengupta@ce.berkeley.edu, {eloi,jiangchuan,rhansen,ryan.swick,dvizzini}@berkeley.edu
‡Department of Mechanical Engineering, University of California, Berkeley
Email: haoc@berkeley.edu

***Abstract*—We take the paradigm of cloud computing developed in the cyber-world and put it into the physical world to create a cyber-physical computing cloud. A server in this cloud moves in space making it a vehicle with physical constraints. Such vehicles also have sensors and actuators elevating mobile sensor networks from a deployment to a service. Possible hosts include cars, planes, people with smartphones, and emerging robots like unmanned aerial vehicles or drifters. We extend the notion of a virtual machine with a virtual speed and call it a virtual vehicle, which travels through space by being bound to real vehicles and by migrating from one real vehicle to another in a manner called cyber-mobility. We discuss some of the challenges and envisioned solutions, and describe our prototype implementation.**

I. Introduction

The key innovations in cyber-physical cloud computing (CPCC) [1] are to have servers move in space and carry sensors and/or actuators. Like regular cloud computing, CPCC customers get a virtual machine (VM) running on a real server. Since a CPCC server moves in space, so does a CPCC VM. Hence we call it a virtual vehicle (VV). The CPCC server is called a real vehicle (RV). By leveraging virtualization technology [2] and mobile agent research [3], [4], we can also have VVs hop or migrate over a network from one RV to another. This means virtual vehicles have two kinds of mobility: a small-time-scale hop, we call *cyber-mobility*, and a larger-time-scale motion with the real vehicle called *physical mobility*. When a VV is bound to an RV, the VV exhibits physical mobility. When it migrates it exhibits cyber-mobility. CPCC, just like regular cloud computing, is meant to work at scale, i.e., at least for tens of vehicle providers, hundreds of real vehicles, and potentially thousands of virtual vehicles.

Since a virtual vehicle moves in space, we envision a programming model for virtual vehicles that exposes their location. This is one implication of making servers move in space. We intend writing programs like "`do c every x units of space`" where c is some computation permitted on usual cloud VMs. Adding sensing and actuation to this suggests a powerful programming model for sampling problems in time and space. The innovation in the programming model is to make space a first-class concept. This makes it different from hybrid systems or embedded computing where only time is a first-class concept [5]. Our approach to programming model research for CPCC will be to add space to models in embedded computing. The second implication of CPCC servers moving in space is *logical mobility*. Since CPU, memory, and I/O are all virtualized, we propose virtualizing space and time as well. In a CPCC programming model it should be perfectly legitimate to program "`do c at location x`" when there exists no machine at location x. This is to be made meaningful by the runtime system using physical mobility and cyber-mobility to move a real vehicle to x and hosting the virtual vehicle holding the program on it.

Sensor networks, when organized as cyber-physical cloud, change from a deployment to a service. For example, a virtual vehicle can be seen as an Amazon Elastic Compute Cloud (E2C) unit [6] plus a virtual speed. If one has contracted a VV that accepts a program like "`do c every 10m and 1s`" then one must have contracted for a VV with a speed in excess of 10 m/s. Whether the cloud realizes this 10-m/s virtual vehicle using ten 1-m/s real vehicles or realizes two 10-m/s virtual vehicles using one 50-m/s real vehicle should be a matter of no concern to the programmer. This is the meaning of virtualizing speed (or space and time). The abstraction resembles the Logical Execution Time (LET) [7] model where time is a specification with semantics. Whether the runtime system implements it or not is a question to be evaluated or proven. The important part is to have a specification that can derive a behavior from the program that is also measurable on the runtime system executing the program.

Next we discuss the binding and migration problem of CPCC through a candidate solution (Section II), and then describe our prototype implementation of a CPCC infrastructure (Section III), followed by a summary of current and future work (Section IV).

This work has been supported by the National Science Foundation (CNS1136141), the European Commission (ArtistDesign NoE on Embedded Systems Design, 214373), the Austrian Science Fund (RiSE NFN on Rigorous Systems Engineering, S11404-N23), the Fundação para a Ciência e Tecnologia (SFRH/BD/43596/2008), and the National Science Foundation (CNS1136141)

II. Binding and Migration

We assume a virtual vehicle is a Turing-equivalent machine with a virtual speed. Since such a machine has a location in space, or even motion in space, if c denotes some computation, we envisage programs such as "`do c every x units of space & t units of time`" where both space x and time t are logical as previously discussed. A semantics would then specify the behavior of such programs as a flow

$$\langle c_0, x_0, t_0 \rangle \to \langle c_1, x_1, t_1 \rangle \to \langle c_2, x_2, t_2 \rangle \to ...$$

where the c_i denote computations specified by programs on the virtual vehicle, x_i the specified location of the $i-$th computation and t_i its specified execution time. The virtual speed of the virtual vehicle relates to this flow as a constraint. If the virtual speed is v m/s then we might impose

$$\frac{X_{i+1} - X_i}{T_{i+1} - T_i} < v$$

as a condition for the behavior to be in conformance with its virtual vehicle. The development of the semantics may accompany the development of the programming model left to the future. The overall problem consists of how to instantiate a runtime system of real vehicles that meets the behavior specified by the virtual vehicles.

Figure 1 illustrates our envisioned design. The (blue) dots denote real vehicles and the (orange) discs denote computations emanating from the various virtual vehicles, i.e., the tuples $\langle c, x, t \rangle$. The runtime system needs algorithms to bind each virtual vehicle to a real vehicle and to determine when to change the binding and migrate the virtual vehicle.

To solve the binding problem illustrated in Figure 1 we make an assumption. If t_i and t_{i+1} are the times of the $i-$th and $(i+1)-$th computations emanating from the virtual vehicle, and t is the current time, then the tuples $\langle c_i, x_i, t_i \rangle$ and $\langle c_{i+1}, x_{i+1}, t_{i+1} \rangle$ are both known to the runtime system at time t. In other words, the logical time and space of the $(i+1)-$th computation must be announced at the execution time of the $i-$th computation. Next we assume the linear interpolation

$$x(t) = x_i + \frac{x_{i+1} - x_i}{t_{i+1} - t_i}(t - t_i)$$

and use $x(t)$ as the position of the virtual vehicle at every time t in the interval (t_i, t_{i+1}). By giving a virtual vehicle a position in logical space at every time, we can now leverage the idea of Voronoi cells for partitioning the Euclidean space regarding the locations of real vehicles. Given a set of locations (in this case, the locations of real vehicles) in the Euclidean plane, a Voronoi cell for a given location p corresponds to all the points whose distance to p is not greater than their distance to any other location. The black lines in Figure 1 illustrate the boundaries of the Voronoi cells (also known as Voronoi tessellation) for all real vehicle locations. Our binding algorithm design has the following steps:

- Given a time t, build a probability distribution for the geographic locations of the logical space locations of all computations produced by all virtual vehicles in the system. If the stochastic process is stationary, then the distribution is the same for all t.
- If one has m real vehicles then tessellate the entire operating area of the cloud into m cells by minimizing the continuous multi-median function for m medians. This can be done using techniques in [8].
- Allocate each virtual vehicle to a Voronoi cell based on its logical position and thus to the real vehicle in the Voronoi cell.
- Program each real vehicle with a sequencing algorithm used to determine the sequence in which the real vehicle will travel through the $\langle c, x, t \rangle$ tuples presented to it by the various virtual vehicles bound to it. This is illustrated by the black curve with an arrow in Figure 1.

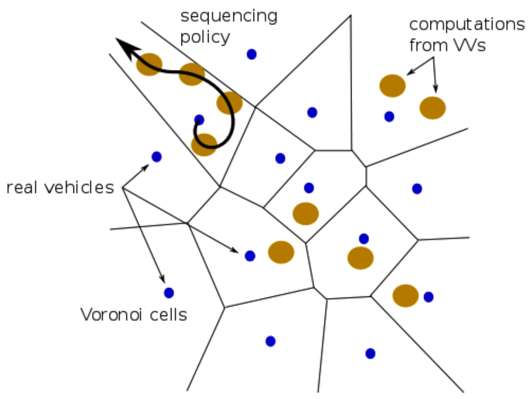

Fig. 1. Binding of virtual vehicles to real vehicles.

Common sequencing algorithms in the literature include "first-come, first-served", the "nearest task policy" [9], or optimal solutions to the traveling salesman problem [8], [9]. The latter is more desirable, though it is NP-hard. For approximation algorithms see [10]. Our own contributions are approximation algorithms for the multiple-vehicle TSP [11].

III. Infrastructure

We have developed a virtualization infrastructure for CPCC called Tiptoe [12] based on the Xen hypervisor [2]. Tiptoe runs bare-metal on Intel hardware and has been tested on a quadcore 19-inch rack server machine as well as on a dualcore Pico-ITX embedded computer (which weighs around 150 grams including WLAN and SSD).

The Xen hypervisor implements a so-called virtual machine monitor (VMM), which (para-)virtualizes the underlying hardware (computer) into so-called domains, or virtual machines, of which each appears as an (almost) exact copy of the hardware [2]. The virtual machine monitor only implements basic (non-real-time) domain scheduling services as well as domain memory and I/O isolation. There is one privileged domain and a dynamic number of unprivileged domains, which may run any (almost) unmodified systems software, if it runs on the underlying hardware without the virtual machine monitor, such as Linux or Windows. The privileged domain runs

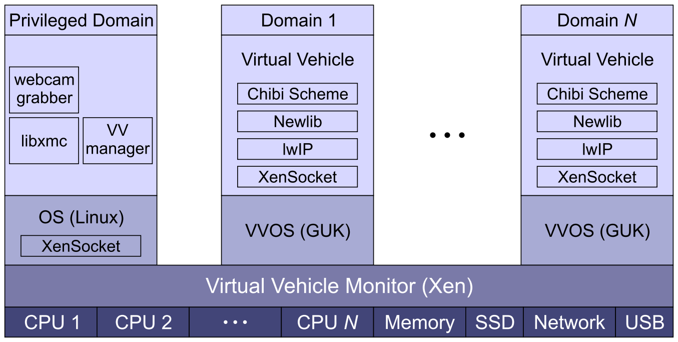

Fig. 2. Schematic overview of Tiptoe.

Linux and serves two important and distinct purposes: domain management and device abstraction. Domain management includes creating, monitoring, and destroying unprivileged domains. Device abstraction is performed by running device drivers exclusively in the privileged domain, and not in any unprivileged domain or in the virtual machine monitor. An unprivileged domain that wishes to communicate with a device may only do so through a virtualized version of the device, which is connected through the virtual machine monitor to the actual device driver running in the privileged domain [2].

Tiptoe enhances Xen in three distinguished categories: (1) domain scheduling through a hybrid EDF-credit scheduler for mixed real-time and non-real-time workloads [12], (2) sensor virtualization through high-bandwidth sensor data multicast for efficient sensor data distribution, and (3) domain migration through runtime-level snapshotting and domain prebooting for low-latency, low-overhead migration performance. The work on multicast (2) and migration (3) is new.

Tiptoe implements what we call a virtual vehicle monitor (VVM), which virtualizes the underlying hardware (sensors, computer, storage, network, actuators) into virtual vehicles of which each appears as an abstract version of the hardware, as opposed to domains, which are (almost) exact copies of the underlying hardware. A virtual vehicle is a domain that essentially runs a light-weight, single-address-space operating system with a Scheme interpreter on top. Figure 2 provides a schematic overview of Tiptoe.

A. *Sensor Virtualization*

Our goal is to enable multiple virtual vehicles to access multiple, possibly high-bandwidth sensors of the underlying hardware at the same time without overloading any processing elements. The problem is that, by the very nature of virtualization, domains and virtual vehicles in particular are isolated from each other in memory and I/O as well as from the underlying hardware. Moreover, virtual vehicles change over time, i.e., they are created, executed, migrated, and destroyed dynamically at runtime.

Our solution follows the same path that Xen has already taken for storage and network I/O with additional support of device-to-domain multicast functionality. Given a sensor device, the appropriate Linux device driver is installed and executed in the privileged domain. A daemon called eyed which we developed from scratch processes in the user space of the privileged domain the video feed obtained through the V4L2 video capture API for Linux [13]. The daemon can handle multiple sensors of the same device type (e.g., front- and rear-facing cameras) and is designed in a way that its process management code is largely decoupled from the actual frame capturing logic. This separation facilitates the reuse of the management code for other sensor devices. Using the so-called libxmc library, which we also developed from scratch, the daemon maintains sensor-to-vehicle mappings to keep track of which virtual vehicle is interested in receiving data from which sensors, and distributes sensor data to the possibly changing set of virtual vehicles. Control data and actual sensor data is communicated through so-called XenStore storage [14] and so-called XenSocket connections [15], respectively.

The actual multicast functionality based on XenSocket is implemented in our libxmc library. Each sensor-to-vehicle mapping is represented by a XenSocket connection. Whenever data is available from a given sensor, the eyed daemon forwards the data to all XenSocket connections for that sensor. The connected virtual vehicles may then asynchronously receive the data. In our experiments, we have already been able to multicast a 300-KB/s video feed to three virtual vehicles hosted on the same server, incurring negligible CPU utilization.

B. *Virtual Vehicle Migration*

Our goal is to migrate virtual vehicles with low latency and low overhead on wireless networks. Migration should be so cheap that, whenever beneficial, it may even become the rule rather than the exception while leaving ample bandwidth for other uses such as video streaming. The problem is, however, that the existing migration facilities in Xen only support migration on domain level where all of a domain's memory content is transferred at least once per migration regardless of any runtime-level information. Migration in Xen transfers a domain's memory content while the domain is still running. For correctness the memory content that has changed during the last transfer is re-transmitted until the point in time when the memory content changes faster in between two transfers than the available transfer bandwidth. At an efficiently computable approximation of that point, the domain is suspended, the changed memory content is transferred once more, and finally domain execution is resumed on the target machine. This method works for any software running in a domain including whole operating systems and provides low latency in the sense that actual domain downtime may be in the order of a few milliseconds for software that exhibits locality in memory access behavior. However, total migration time is at least proportional to a domain's memory size which may be in the order of MBs and even GBs resulting in high bandwidth requirements [16]. The bandwidth requirements may be somewhat reduced by applying memory compression algorithms on migrating domains [17], which is nevertheless still not sufficient for our purposes.

Our solution is based on runtime-level snapshotting as well as on domain pre-booting. On each server, the previously mentioned virtual vehicle manager maintains a number of pre-booted and then suspended domains as targets for virtual vehicle migration. Upon migrating a virtual vehicle, its domain is immediately suspended to snapshot the vehicle state whose size is in the order of a few KBs in our current setup. This includes virtual vehicle data as well as the state of the vehicle's network stack. The state is then transferred to a pre-booted domain on the target server. Finally, the pre-booted domain completes its boot process, advances to the received state, and then resumes the execution of the migrated virtual vehicle. The result is low-latency and low-overhead migration performance. Snapshotting at the language runtime level significantly reduces the bandwidth requirements of Xen migration but ties the use of the migration facility to the language runtime; it is not possible to migrate arbitrary binary code, as with Xen's native migration, but only code written in a high-level programming language extended with support for migration. Domain pre-booting is an optimization of runtime-level snapshotting to further reduce latency.

Hosting a scalable number of virtual vehicles requires domains with small memory footprint. We therefore chose the GUK microkernel [18] as foundation of a virtual vehicle operating system (VVOS). GUK is a single-address-space, lightweight microkernel originally designed to support a JVM running inside a Xen domain. GUK implements basic thread scheduling and device drivers for virtual I/O devices only. A virtual I/O device provides a well-defined interface that abstracts the low-level details of the underlying real I/O device, which is only accessible to the privileged domain. Any access to a virtual I/O device is routed to the privileged domain which then accesses the real I/O device on behalf of the connected domain. Moreover, since multiple domains may share a single I/O device, the privileged domain performs I/O scheduling for all incoming I/O requests from all domains.

Virtual vehicles need network access to migrate but also to relay sensor data to the ground station. We integrated the lwIP library [19] into GUK, providing TCP and UDP connectivity to domains hosting virtual vehicles. We extended lwIP to allow for the migration of live TCP and UDP connections, i.e., live TCP and UDP connections are kept alive across any number of migrations. For prototyping virtual vehicle behavior, we integrated Chibi Scheme [20] into our software stack. Chibi is an implementation of the Scheme programming language, specifically of R5RS [21]. We supplemented Chibi's libraries to expose the C APIs offered by the platform, such as lwIP, to programs written in Scheme.

IV. Current and Future Work

We aim at devising models, algorithms, and protocols for solving the binding and migration problem of CPCC as well as developing a diverse testbed for CPCC (in simulation and for real) that includes several types of hosts such as cars, buses, people with smartphones, and unmanned aerial vehicles (UAVs). So far we developed a low-cost lightweight Flying Wing UAV based on the Zephyr airframe, and the JAviator—a high-performance quadrotor UAV built from scratch [22]. The UAVs are equipped with an autopilot and will carry a computational platform for CPCC such as the previously mentioned Pico-ITX board. We plan to equip the UAVs with sensors ranging from CO_2 concentration sensors to EO/IR cameras. We have also started working on an Android port of our virtualization infrastructure so that virtual vehicles may seamlessly migrate across UAVs and smartphones.

References

[1] S. Craciunas, A. Haas, C. Kirsch, H. Payer, H. Röck, A. Rottmann, A. Sokolova, R. Trummer, J. Love, and R. Sengupta, "Information-acquisition-as-a-service for cyber-physical cloud computing," in *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. Symposium on Operat. Syst. Princ.* ACM, 2003, pp. 164–177.

[3] J. E. White, "Mobile agents, software agents," MIT, Tech. Rep., 1997.

[4] D. B. Lange and M. Oshima, "Seven good reasons for mobile agents," *Commun. ACM*, vol. 42, pp. 88–89, March 1999.

[5] C. Kirsch and R. Sengupta, *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007, ch. The Evolution of Real-Time Programming.

[6] http://aws.amazon.com/ec2/.

[7] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, January 2003.

[8] M. Pavone, E. Frazzoli, and F. Bullo, "Adaptive and distributed algorithms for vehicle routing in a stochastic and dynamic environment," vol. 56, no. 6, pp. 1259–1274, 2011.

[9] D. Bertsimas and G. van Ryzin, "Stochastic and dynamic vehicle routing with general inter-arrival and service time distribution," *Advances in Applied Probability*, 1991.

[10] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Dover Publications, 1998.

[11] S. Rathinam and R. Sengupta, "3/2-approximation algorithm for two variants of a 2-depot hamiltonian path problem," *Oper. Res. Lett.*, pp. 63–68, 2010.

[12] S. Craciunas, C. Kirsch, H. Payer, H. Röck, and A. Sokolova, "Programmable temporal isolation in real-time and embedded execution environments," in *Proc. Workshop on Isolation and Integration in Embedded Systems (IIES)*. ACM, 2009.

[13] "The linuxtv project," 2011. [Online]. Available: http://linuxtv.org

[14] Citrix Systems Inc., "XenStore," March 2011, http://wiki.xensource.com/xenwiki/XenStore.

[15] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin, "XenSocket: A high-throughput interdomain transport for virtual machines," in *Proc. ACM/IFIP/USENIX 2007 International Conference on Middleware*, ser. Middleware '07. Springer, 2007, pp. 184–203.

[16] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005, pp. 273–286.

[17] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," *2009 IEEE International Conference on Cluster Computing and Workshops*, pp. 1–10, 2009.

[18] Oracle Inc., "GUK," March 2011, http://labs.oracle.com/projects/guestvm/shared/guk/index.html.

[19] A. Dunkels, "Minimal TCP/IP implementation with proxy support," Master's thesis, SICS, February 2001.

[20] A. Shinn, "Chibi-Scheme - Small Footprint Scheme for use as a C Extension Language," January 2011, http://code.google.com/p/chibi-scheme/.

[21] R. Kelsey, W. Clinger, J. Rees, H. Abelson, R. Dybvig, C. Haynes, G. Rozas, D. Bartley, R. Halstead, D. Oxley, G. Sussman, G. Brooks, C. Hanson, K. Pitman, and M. Wand, "Revised 5th report on the algorithmic language Scheme," *ACM SIGPLAN Notices*, vol. 33, pp. 26–76, 1998.

[22] S. Craciunas, C. Kirsch, H. Röck, and R. Trummer, "The JAviator: A high-payload quadrotor UAV with high-level programming capabilities," in *Proc. AIAA Guidance, Navigation and Control Conference*, 2008.