CLOUD COMPUTING ON WINGS: APPLICATIONS TO AIR QUALITY

Sengupta, R.*, Hansen, R.*, Pereira, E.*, Huang, J.*, Kirsch, C.†, Chen, H.‡,
Landolt, F.†, Lippautz, M.†, Rottmann, A.†, Swick, R.*,
Trummer, R.† and Vizzini, D.*

We extend the concepts from cloud computing developed in the cyber–world into the physical world, creating a new paradigm called cyber–physical computing cloud. Under the proposed framework, a cyber–physical server is allowed to move in space and perform real–world interactions such as sensing and actuation. The analog for cloud computing's virtual machine, under cyber–physical cloud computing, is the "virtual vehicle". Thus, cyber–physical servers must be hosted by real vehicles, such as automobiles, planes, smartphones, and unmanned air systems. A likely candidate for the application of cyber–physical cloud computing is in air quality monitoring; a collaborative network of vehicle–mounted gas sensors would make possible unique types of atmospheric sensing in three dimensions plus time. We discuss the challenges involved in establishing systems according to this new paradigm, and provide our envisioned solutions, and go on to describe specific applications in air quality and our prototype implementation.

INTRODUCTION

The key innovations in cyber-physical cloud computing (CPCC) [1] are to have servers move in space and carry sensors and/or actuators. As with 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 hardware of a CPCC server is called a real vehicle (RV). By leveraging virtualization technology [2] and mobile agent research [3, 4], we can also have VVs migrate over a network from one RV to another. This means virtual vehicles have two modes of mobility: a short time–scale hop facilitated by network migration and called *cyber-mobility*, and a long time–scale mobility facilitated by RVs called *physical mobility*. CPCC, like regular cloud computing, is designed to work at scale, i.e. for at least tens of vehicle providers, hundreds of real vehicles, or potentially thousands of virtual vehicles.

Since a virtual vehicle is allowed to move in space, we envision a programming model for virtual vehicles that expresses their location. CPCC aims to allow 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 computation suggests a powerful programming model for environmental sampling problems in time and space. The innovation in the programming model is to make space a first-class concept. This makes CPCC different from hybrid systems and embedded computing where time alone 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

*Department of Civil and Environmental Engineering, University of California, Berkeley, Emails: sengupta@ce.berkeley.edu, {rhansen,eloi,jiangchuan,ryan.swick,dvizzini}@berkeley.edu

†Department of Computer Sciences, University of Salzburg, Emails: {christoph.kirsch, florian.landolt, michael.lippautz, andreas.rottmann, rainer.trummer}@cs.uni-salzburg.at

‡Department of Mechanical Engineering, University of California, Berkeley, Email: haoc@berkeley.edu

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.

The type of sensor network made possible by CPCC will provide an excellent platform for environmental sensing and, in particular, air quality monitoring. We will present how we envision CPCC to be utilized in this specific domain with two examples of sensing problems solved uniquely by a collaborative network of virtual vehicles. We will also compare the proposed solution to existing sensor deployments and discuss how a mobile air–based network can be integrated with current systems for improved air quality modeling and forecasting.

Next we discuss the binding and migration problem of CPCC through a candidate solution (Section "BINDING AND MIGRATION"), then describe our prototype implementation of a CPCC infrastructure (Section "INFRASTRUCTURE"). In section "CPCC IN AIR QUALITY APPLICATIONS" we discuss the use of CPCC for air quality applications. We proceed with a description of our testbed (Section "CPCC TESTBED") followed by a summary of current and future work (Section "CONCLUSIONS AND FUTURE WORK").

BINDING AND MIGRATION

In this section we address the binding and migration problem. We start by presenting informally the semantics of our programming model. We then proceed by sketching an possible algorithm for solving the problem.

Semantics

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. Let $\langle c, x, t\rangle$ denote a configuration, i.e. a snapshot of the current state of the computation. A semantics would then specify the behavior of such programs as a sequence of configurations:

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

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

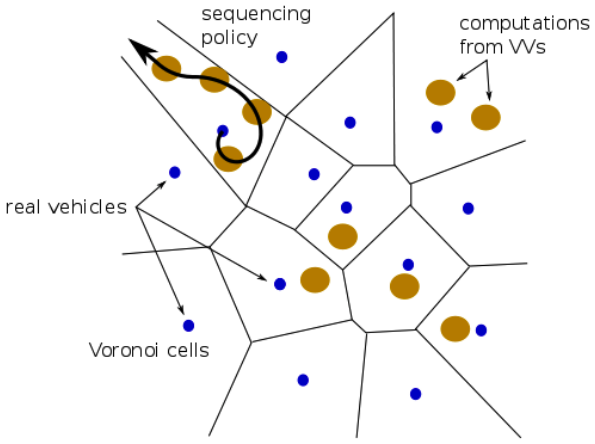

Figure 1: Binding of virtual vehicles to real vehicles.

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. X_i and T_i are random variables denoting the realized physical location of the i-th computation and execution time. 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.

Algorithms

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].

Once this allocation is done the virtual vehicle has a motion in both logical and physical space. If the first is

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

and the second

$$\langle c_0, x'_0, t'_0\rangle \to \langle c_1, x'_1, t'_1\rangle \to \langle c_2, x'_2, t'_2\rangle \to ...$$

then the error is

$$\|((x_0, t_0), (x_1, t_1), \ldots) - ((x'_0, t'_0), (x'_1, t'_1), \ldots))\|,$$

where $\|\cdot\|$ denotes an appropriate norm. The protocols for binding virtual to real vehicles should be designed to minimize the norm. Observe that between t_i and t_{i+1} there may be a discrepancy between the logical position of the virtual vehicle obtained by linear interpolation and the physical position of the real vehicle hosting it. We do not count this discrepancy in our cost function. Nevertheless, this discrepancy at intermediate times is a useful signal for feedback control working to keep the real vehicle moving on schedule to its next logical position.

Note the Voronoi tessellation at time t is not generated using the positions of the real vehicle at time t. Rather, it is generated using the probability distribution described above and the number of real vehicles in the system. Thus, as long as the first is stationary and the number of real vehicles constant, the tessellation is time-invariant. This design choice leads to the phenomenon of cyber-mobility. In a behavior

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

it is possible x_i and x_{i+1} will lie in different Voronoi cells, as the tessellation is time invariant. In this case our design hops the virtual vehicle from the real vehicle in the first cell to the one in the second cell. The process is called migration and the algorithm determining migration is driven by the Voronoi tessellation computed for the cloud.

Permitting cyber-mobility via a virtual machine migration mechanism allows us to reduce the physical mobility demand by using cyber-mobility. Real vehicles can stay in their own Voronoi cells while, the virtual vehicles can migrate if they are "too mobile" for their real vehicle hosts. The price of this is some network bandwidth and the migration delay.

We choose this algorithm design for binding and migration because it is supported by optimality results in queuing theory or more precisely a geographical extension of queuing theory since the errors occur in time and space [9, 12], [8]. To use this literature

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

is assumed to be a random process, that is further required to be at least wide-sense stationary, i.e. at least the first two moments do not vary with respect to time and space. The objective function to be optimized is then formulated as the expected value $E[\|(X,T)-(X',T')\|]$ or the standard deviation $E[\|((X,T)-(X',T')-E[(X,T)-(X',T')]\|)^2]$. This two dimensional cost is reduced to one dimension by treating the X_i as constraints, meaning $X = X'$ and

$$E[\|(X,T)-(X',T')\|] = E[\|T-T'\|].$$

The objective is then to minimize the expected value of the time spent by a computation in the system T_S, i.e., the difference in between the time at which the computation is first presented to the real vehicle and the time at which it is executed. This objective function relates to ours in the following way. The $i-$th computation from a virtual vehicle is executed in a timely fashion if $A_i + T_{Si} \leq T_i$, where A_i is the arrival time of the i-th task. The probability a computation will be late is determined by A_i, T_i, which are known to the system, and the distribution of T_{Si}.

When the inter-arrival time $A_i - A_{i-1}$ is i.i.d. and exponentially distributed and the locations X uniformly distributed over the entire operating area of the cloud, one can prove using results in [9] that the expected value of the time spent by a computation in the system T_S, is bounded below by

$$\bar{T}^* \geq \frac{1}{v} H^*_m(\mathcal{Q}) + \bar{s}, \qquad \varrho \rightarrow 0^+ \tag{1}$$

where $\bar{T}^*$ is the minimum of T_S under all policies, m is the number of vehicles, $\mathcal{Q} \subset \mathbb{R}^d$ is a compact set, $P = (p_1, \ldots, p_m)$ are the m points in $\mathcal{Q}$. $H^*_m(\mathcal{Q}) = H_m(P^*_m(\mathcal{Q}), \mathcal{Q})$, $P^*_m(\mathcal{Q}) = \arg\min_{P \in Q^m} H_m(P, Q)$, $H_m(P, \mathcal{Q}) \doteq \mathbb{E}\left[\min_{k\in\{1,\ldots,m\}} \| p_k - x \|\right]$, and by

$$\bar{T}^* \geq \frac{\beta^2_{TSP,2}}{2} \frac{\lambda[\int_{\mathcal{Q}} f^{1/2}(x)dx]^2}{m^2 v^2 (1-\varrho)^2}, \qquad \varrho \rightarrow 1^- \tag{2}$$

where λ is the Poisson arrival rate, $f(x)$ is the geographical distribution of tasks, and v is the speed of vehicles. The problem is known in the logistics literature as the $m-$Dynamic Traveling Repairman Problem (DTRP). In the light load case ($\rho \rightarrow 0^+$) a sequencing policy executing the computation nearest in location to the current one is optimal. In the heavy load case ($\rho \rightarrow 1^-$) it is asymptotically optimal for the real vehicle to follow a divide and conquer traveling salesman tour of the computations in its cell [8].

The $\langle c, x, t\rangle$ arrival process in the geographical queuing literature is an extension of the Poisson arrival model in queuing theory with a space dimension. Usually, the x in each job $\langle c, x, t\rangle$ is chosen from a uniform distribution over the entire operating area containing all the real vehicles after choosing the t from a distribution such as the memoryless exponential inter-arrival time one. However, the virtual vehicle abstraction viewed in queuing-theoretic terms suggests a model we have not found thus far in the literature. In geographical queuing if T_i and X_i are the desired execution time and location of computation C_i, and T_{i+1} and X_{i+1} that of computation C_{i+1}, then the probability distribution over $X_{i+1} - X_i$ is independent of the value of the difference $T_{i+1} - T_i$. Typically, $T_{i+1} - T_i$. is Poisson with some rate parameter λ_T and the sequence X_i is Poisson with another rate parameter λ_X. The virtual vehicle abstraction is

better modeled by assuming the distribution of the difference $X_{i+1} - X_i$ should depend upon the value of the difference $T_{i+1} - T_i$.

For example, if one has purchased a 20 m/s virtual vehicle and $T_{i+1} - T_i$. is one second, then perhaps X_{i+1} should lie within the 20 meter circle about X_i with probability one. If $T_{i+1} - T_i$. is two seconds, then perhaps X_{i+1} should lie within the 40 meter circle about X_i with probability one. One may note that such a model is only one of the many ways of introducing dependence. Another way might be to think of the 20 m/s as an expected value and assume that $\|X_{i+1} - X_i\|$ is distributed with expected value 20 meters if $T_{i+1} - T_i$. is one and with expected value 40 meters if $T_{i+1} - T_i$. is two, and so on. Such a model is closer to source rate constraints in the ATM networking literature on quality of service [13]. Such a virtual vehicle is allowed to burst in the same sense as the QoS networking literature if it has been beneath its rate in the past or will be in the future. Intuition suggests this dependence of the space differences on time differences should lead to better policies. When tasks are generated closely in time making it harder for the real vehicles, the dependence dictates they must also be clustered closer in space, making it easier for the real vehicles, and vice versa. In the usual geographic queuing model X_{i+1} is just as likely to be far from X_i as near it in a manner independent of $T_{i+1} - T_i$. The vehicle in the geographic queuing model can jump around more. This is an advantage of the virtual vehicle abstraction. The actual $E[T_S]$ achieved by virtual vehicles should be smaller than the one we have analyzed here using the geographic queuing literature.

INFRASTRUCTURE

We have developed a virtualization infrastructure for CPCC called Tiptoe [14] 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 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 [14], (2) sensor virtualization through high-bandwidth sensor data multicast for efficient sensor data distribution, and (3) domain migration through runtime-level snapshotting and domain pre-booting 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

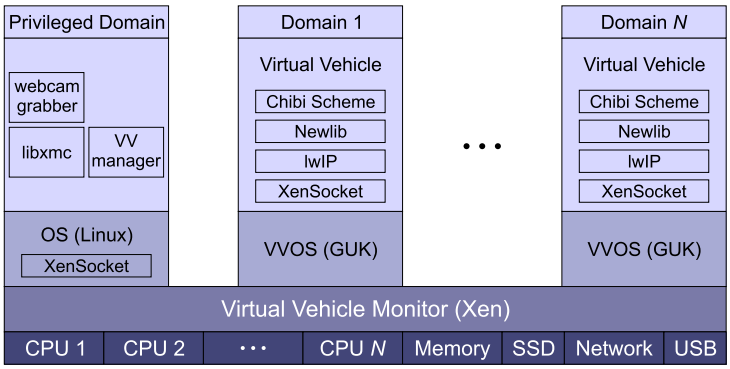

Figure 2: Schematic overview of Tiptoe.

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.

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 [15]. The daemon can handle multiple sensors of the same device type (e.g., CO_2 and methane concentration sensors) 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 [16] and so-called XenSocket connections [17], respectively.

XenStore [16] is an information storage space which is part of Xen. It can be thought of as a hierarchical data structure, similar to a device tree, that is shared between domains. XenStore is used for device discovery and for storing device metadata. A virtual vehicle communicates its interest in sensors to the eyed daemon using our libxmc library, which, in turn, implements the necessary functionality for using XenStore (and XenSocket) services. The daemon advertises the available sensors via XenStore for all virtual vehicles to see. The virtual vehicles that have expressed interest in a given sensor will then receive its sensor data from the daemon through dedicated XenSocket connections.

XenSocket [17] is a high-throughput inter-domain communication infrastructure. Note that

the Xen hypervisor has been designed with minimalism in mind to keep the hypervisor as simple and small as possible, no general-purpose inter-domain communication mechanism is included. Although Xen's virtualized networking infrastructure could be used to realize inter-domain communication, the provided throughput falls far short of the bandwidth needed to transmit large amounts of data such as video feeds. In contrast, XenSocket may provide up to 72 times the throughput of a standard Xen domain-to-domain TCP stream [17].

XenSocket enables two domains to communicate asynchronously via shared memory by establishing an unidirectional communication link between the two domains. The shared memory is allocated at the OS level and made accessible to user-level applications through a socket-like API. Only minimal interaction with the hypervisor is required. Once the memory region is mapped to the address spaces of both domains, the per-message call overhead is in our implementation one system and one hyper call in the privileged domain plus one hyper call in the receiving domain. The message size is currently preset to 128 KB. Since the code base of the XenSocket project was outdated, we forward-ported it to current versions of Xen (4.0.1) and Linux (3.0).

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.

The CPU load imposed by our systems is moderate. We determined the daemons CPU utilization using sysstat. The CPU load was ranging between 0 and 3 %. According to xentop, CPU utilization of each VV was beneath 0.5 %.

Virtual Vehicle Migration

Our goal is to migrate virtual vehicles with low latency and low overhead on wireless networks. This provides advantages beyond a physical network such as continuous sensing during prolonged virtual flights in spite of relatively low–endurance planes entering and leaving a region of interest. 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 imaging 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 on 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 [18]. The bandwidth requirements may be somewhat reduced by applying memory compression algorithms on migrating domains [19], 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 [20] 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.

We have combined GUK with the Newlib C library [21] to be able to run C applications and libraries designed for POSIX-like operating systems with little porting effort. To allow applications access to static data, we extended Newlib to support a simple, read-only file system that can be linked into the image of a domain that hosts a virtual vehicle. Additionally, we incorporated the memory allocator of the GNU C library into our system, as we ran into fragmentation issues with the naive allocator provided by GUK when exposed to applications via Newlib.

Virtual vehicles need network access to migrate but also to relay sensor data to the ground station. We integrated the lwIP library [22] 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 [23] into our software stack. Chibi is an implementation of the Scheme programming language, specifically of R5RS [24]. We supplemented Chibi's libraries to expose the C APIs offered by the platform, such as lwIP, to programs written in Scheme.

Upon migration, a virtual vehicle establishes a TCP connection to the virtual vehicle manager of the target server, then sends a representation of its Scheme continuation over that connection, and finally terminates after its continuation has been successfully transmitted. After the virtual vehicle manager has received the representation of the continuation, it copies the data constituting this representation into the address space of a pre-booted domain. After the copying process has completed, the manager signals the domain to reinstate the continuation and then continue the execution of the virtual vehicle.

CPCC IN AIR QUALITY APPLICATIONS

The paradigm established by CPCC is well–suited to problems of air quality management and atmospheric pollution detection. It is expected that, as global urbanization and climate change continue, it will become increasingly necessary to sense, understand, predict, and control airborne pollutants, including greenhouse gases (GHG), primary and secondary aerosols and precursors, and trace gases [25, 26]. Among the critical measurements needed for successful and accurate pollution budgeting is terrestrial and biotic gas flux, with species of interest including CO_2, water, methane, and isoprene. Currently, such measurements are carried out in the limited set of locations where researchers have set up permanent or semi–permanent stations; airborne sensing is limited to short, isolated flights in large manned research vessels [27].

Comprehensive understanding of air pollution demands ubiquitous sensing, which in turn requires the integration of a multitude of sensing platforms. It is infeasible for permanent flux towers to be constructed in every region of interest, due to maintenance considerations, concerns over the safety of the site, property issues, and myriad other reasons. Likewise, airborne sensing, in its current form, is of limited use—infrequent research flights may be used for such purposes as verification and calibration of satellite imagery [28], but there is much valuable data to be gained toward improved gas and aerosol budgets, and more effective pollution control, through the facility of a real–time network of controlled, mobile atmospheric sensors. Such sensors could provide data where geographical gaps exist between stationary sensors and, perhaps more importantly, provide in–situ data that follows the phenomena it represents. An active Lagrangian network [29] of airborne sensors could reveal the dynamics of otherwise unobserved reactions such as secondary aerosol formation [26].

Air Sampling Example Cases

Consider two potential unique implementations of CPCC applied toward air quality monitoring. First, an experiment requires periodic trace gas concentration sampling at a single point in space, repeated at a certain frequency for a specified duration. This is the type of sensing carried out by ground–based devices such as CO_2 flux towers [30], but in certain cases structurally supported sensors may be infeasible. For example, high altitude phenomena such as chemical fluxes across the tropopause cannot be directly measured from the ground. Using the CPCC paradigm, a programmer could write a program "`do c at x`", with `x` evaluating to a single spatial coordinate, i.e. latitude, longitude, and altitude at each time for which data is requested, and `c` meaning "measure and report". The cloud is then responsible for performing the requested sampling and computation at the defined point however it can with the available VVs.

As a second example, assume one or more chemical tracers is distributed through a parcel of air. A programmer in this instance is interested in maintaining the best possible estimate of the entire distribution through the volume. Thus, `x` as provided to the CPCC interface would define a finite volume, rather than a discrete point, and `c`, rather than being a simple command to sense entails a degree of computational complexity involving some atmospheric modeling and forecasting. This type of sensing involves a control problem inherently impossible with static sensor networks but eminently solvable with the mobile sensing network made possible through CPCC.

Additional Benefits

One further contribution to atmospheric studies promised by CPCC is to remedy some characteristic problems associated with RVs, such as finite flight duration and geographic range. Whereas a single plane, particularly a small unmanned vehicle, might be constrained to a mission on the order of hours in the time domain and up to several hundred miles spatially, a CPCC network of vehicles can provide uninterrupted mission service by maintaining a continuous trajectory of a VV by hopping among RVs.

Atmospheric scientists will benefit from this abstraction by gaining a network of sensors with controllable grid resolution, thus capable of resolving phenomena on a wide range of scales. Furthermore, CPCC will allow for significant supplemental to be shared with existing ground–based networks such as FLUXNET [31] while simultaneously giving researchers access to a means of more detailed study in one or multiple locations. Unmanned air vehicles are bound to be an invaluable asset to atmospheric scientists and air quality engineers, and CPCC will allow interested communities to systematically utilize the complex dynamic physical network efficiently and valuably.

CPCC TESTBED

To evaluate the concept of CPCC and the developed protocols and architectures, we propose experiments with RVs in the forms of: cars carrying CO_2, CO, NO_x and SO_x, O_3, and particulate matter, sensors and cameras, people hosting computers with sensors, and two different sets of air vehicles: small flying wings equipped with CO_2 sensors and low cost cameras and JAviator quad-rotors equipped with cameras. The project team has experience building and operating these types of vehicles and sensors.

We aim at developing a testbed for CPCC that includes the following manifestations of real vehicles:

- Private automobiles;
- Smartphones and their owners;
- Public transit vehicles;
- Weather Balloons;
- Flying Wing UAVs;
- JAviator quadrotor UAVs [32].

For the purpose of atmospheric monitoring, each of these RVs can be fitted with sensors for detection of one or more of the following parameters: CO_2, CO, NO_x, SO_x O_3, Water vapor, Temprature, and Multi–spectral imagery. This list certainly could be expanded as resources are availed and research interest broadens, though the feasibility of any given sensing payload is limited by the real vehicle on which it must be affixed. Furthermore, we acknowledge that the motion of some of these hosts may be controllable, others partially controllable, and some uncontrollable. For example, a bus hosting one of our servers will still move by its schedule, i.e., it is completely uncontrollable by the cloud. People walking or driving cars may be partially controllable through the use of a system of incentives, and others such as the UAVs, or automobiles owned by the cloud, may be controllable subject to mode–specific constraints such as wind or traffic congestion. One goal of CPCC is to have the smallest number of dedicated

and controlled vehicles required to complement the uncontrollable hosts in the cloud, as we anticipate the former are costlier than the latter.

Unmanned Air Vehicles

The Flying Wing we have developed for this use is a lightweight, low-cost UAV (see Figure 3(a)). It has a tailless fixed-wing airframe with elevons control surfaces and electric engine. We are currently using two types of Commercial Off-The-Shelf (COTS) airframes: the Zagi and the Zephyr. Either airframe is equipped with an open-source autopilot, Ardupilot Mega [33], that provides waypoint control, meaning given a sequence of GPS waypoints (locations to visit) the autopilot controls the aircraft to meet those waypoints without the need of human intervention. The JAviator is a high-performance quadrotor helicopter UAV built from scratch [32] (see Figure 3(b)).

(a) Flying Wing (b) Javiator

Figure 3: UAVs

The Flying Wing is our best candidate platform for performing air quality sampling. Since the propeller is on the back of the wing, its influence on the sensors measures due to air disturbances is minimized. We also studied the use of a blimp for this kind of mission. Although this is a good solution regarding minimizing disturbances it doesn't satisfy our requirements in terms of speed.

Environmental payload

The sensors currently in our testbed are webcams, CO_2 concentration sensors, temperature, and pressure sensors. We developed a low-cost, self-contained, lightweight sensing payload that can be used by small scale UAVs (see Figure 4). The payload gathers data from several sources, tags each sample with GPS information, and logs the data into an SD card. The overall payload is composed of the following components:

- Arduino Uno micro-controller;
- GPS Logger Shield;
- GPS;
- K30 CO_2 Sensor;
- BMP085 Air Pressure sensor;
- SHT15 Temperature/Humidity sensor;
- 5V NiCd Battery;
- Power/I2C board;

(a) Inner vision (b) Outer vision

Figure 4: Payload

The sensor selected for CO2 detection is the K-30 Sensor [34]. The measurement range is from zero to 10,000 ppm, with an accuracy of +/- 30 ppm and a sensitivity of +/- 20 ppm of the measured value. This resolution is coarse in comparison to ambient CO_2 levels, but will still provide a notion of any greatly increased concentrations on a localized basis. Non-dispersive infrared waveguide technology is used to detect the CO_2 concentration.

Relative humidity and temperature (RH/T) detection is achieved in a combined sensor package, the Sensirion SHT15. This sensor comes pre-calibrated, and includes an amplifier, an Analog to digital converter, and a digital interface. Under typical operating conditions, such as those likely to be encountered in environmental sensing, relative humidity sensing is accurate to within 2%, and temperature within 0.3°C.

A GlobalSet EM-406a GPS unit is used for horizontal tracking and speed estimates. This information provides location tags for every point of data collected by the environmental sensors. The unit provides accuracy to within 5 meters when Wide Area Augmentation System is enabled. A cold start requires 42 seconds, on average, to acquire a satellite lock, while a hot start takes only 8 seconds. The unit is capable of withstanding accelerations up to 4G which should be sufficient to survive typical airplane landings and minor collisions.

While GPS is used for horizontal locating, altitude is determined from barometric pressure. This is accomplished using a Bosch BMP085 digital pressure sensor. This model uses the piezo resistive effect to measure strain across a vacuous vessel. The package comes pre-calibrated and features an internal thermal sensor to compensate for temperature fluctuations. Pressure readings are typically accurate to within 1.0 hPa between 300 and 1100 hPa, and between 0 and 65.

Scenario: Greenhouse gas sampling

The Greenhouse Gas Sampling exercise will be carried out with multiple persons, cars, and aircraft (around 5) that walk, drive, and fly to cover an area around UC Berkeley's Richmond Field Station. We will setup multiple virtual vehicles. Each will have several kinds of sense and act programs. The programs will sample a trace gas (i.e. CO_2, CO, or O_3) and simultaneously monitor a video feed from an on–board camera, raise in alarm if the gas concentration exceeds a set threshold or if a specific pattern is recognized from the image sensor. All or some vehicles will be equipped with the required sensors. By separating the mission of measuring different kinds of gas and taking pictures at different rates and places, we aim to simulate the existence of one cloud instead of multiple mobile sensor networks. This experiment is geared towards providing a low cost and flexible alternative to existing expensive greenhouse gas measurements efforts using fixed towers or large UAVs. The idea behind using different sensors (GHG and Camera)

is to evaluate the system under different task loads, which have different communication and hardware requirements.

Figure 5 presents CO_2 concentration samples collected driving around Berkeley, CA, in May 2011. The data samples are overlapped in Google Maps using a color code: blue (concentration

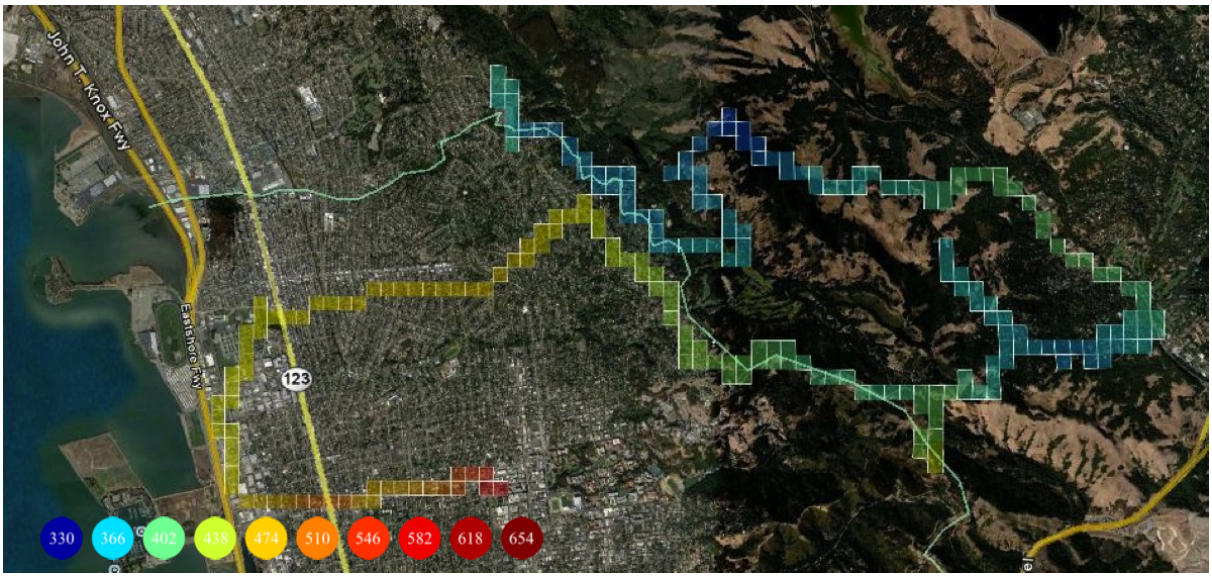

Figure 5: CO_2 concentration samples collected at Berkeley, CA, in May 2011.

of 330 ppm) to brown (concentration of 654 ppm). Figure 6 presents data collected in another experiment using the same CO_2 concentration sensor installed together with a low-cost air pressure and temperature sensor in a tethered weather balloon. The color code now represents

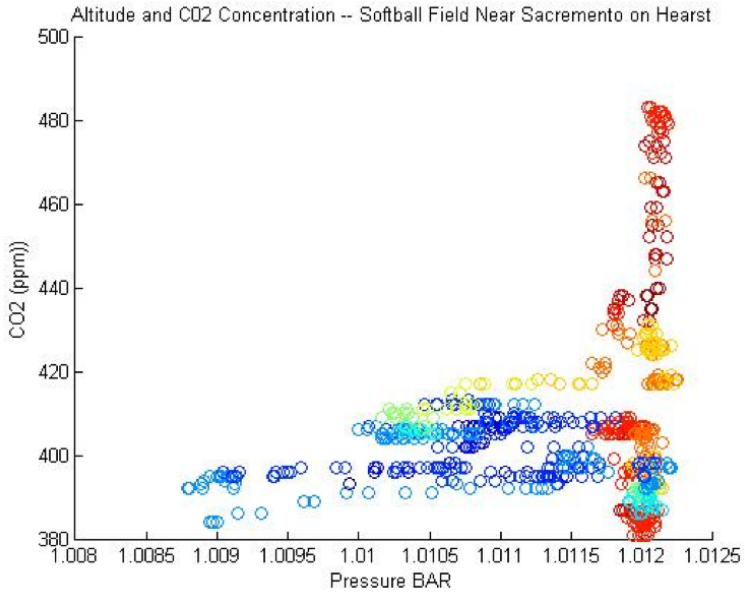

Figure 6: CO_2 concentration, temperature and air pressure samples collected at Berkeley, CA, in May 2011.

the temperature: from blue (16.5° C) to red (29.8° C). Note that at higher air pressure (low altitude) the concentration of CO_2 and the temperature is higher than at lower air pressure (higher altitude). This testing is in preparation for an experiment in which the cloud will host a virtual application able to raise an alarm if a CO_2 threshold is surpassed or take a picture and raise an alert if a possible environmental threat is detected. The application specification will abstract out the real platform. By separating the mission of measuring different kinds of physical quantities and taking pictures at different rates and places, we aim at simulating the idea of one cloud instead of multiple mobile sensor networks.

CONCLUSIONS 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 have developed a low-cost lightweight Flying Wing UAV based on the Zephyr airframe, and the JAviator—a high–performance quadrotor UAV built from scratch [32]. The UAVs are equipped with an autopilot and will carry a computational platform for CPCC such as the aforementioned Pico-ITX board. We plan to equip the UAVs with sensors ranging from CO_2 concentration sensors to EO/IR cameras. A system implementation of an ensemble Kalman filter to control a sensor network for optimal sampling and estimation of a distributed atmospheric chemical such as CO_2 is under development. We have also started working on an Android port of our virtualization infrastructure so that virtual vehicles may seamlessly migrate across UAVs and smartphones.

ACKNOWLEDGEMENT

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).

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] D. Bertsimas and G. Van Ryzin, "A stochastic and dynamic vehicle routing problem in the euclidean plane," MIT, Sloan School of Management, Working papers 3286-91., 1991.

[13] D. McDysan, *QoS and traffic management in IP and ATM networks*. New York, NY, USA: McGraw-Hill, Inc., 2000.

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

[21] J. Johnston, "The Newlib Homepage," January 2011, http://sourceware.org/newlib/.

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

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

[24] 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.

[25] P. Sellers, F. Hall, R. Kelly, A. Black, D. D. Baldocchi, J. Berry, M. Ryan, K. Ranson, P. Crill, D. Lettenmaier *et al.*, "Boreas in 1997: Experiment overview, scientific results, and future directions," *Journal of Geophysical Research*, vol. 102, no. D24, pp. 28 731–28, 1997.

[26] J. Odum, T. Hoffmann, F. Bowman, D. Collins, R. Flagan, and J. Seinfeld, "Gas/particle partitioning and secondary organic aerosol yields," *Environmental Science & Technology*, vol. 30, no. 8, pp. 2580–2585, 1996.

[27] D. D. Baldocchi, "'breathing' of the terrestrial biosphere: Lessons learned from a global network of carbon dioxide flux measurements systems," *Australian Journal of Botany*, vol. 56, pp. 1–26, February 2008.

[28] H. Singh, W. Brune, J. Crawford, F. Flocke, and D. Jacob, "Chemistry and transport of pollution over the gulf of mexico and the pacific: spring 2006 intex–b campaign overview and first results," *Atmospheric Chemistry and Physics*, vol. 9, no. 7, pp. 2301–2318, 2009.

[29] O. Tossavainen, J. Percelay, A. Tinka, Q. Wu, and A. Bayen, "Ensemble kalman filter based state estimation in 2d shallow water equations using lagrangian sensing and state augmentation," December 2008.

[30] D. D. Baldocchi, "Assessing the eddy covariance technique for evaluating carbon dioxide exchange rates of ecosystems: past, present, and future," *Global Change Biology*, vol. 9, pp. 479–492, April 2003.

[31] D. D. Baldocchi, E. Falge, L. Gu, R. Olson, D. Hollinger, S. Running, P. Anthoni, C. Bernhofer, K. Davis, and R. Evans, "Fluxnet: A new tool to study the temporal and spatial variability of ecosystem–scale carbon dioxide, water vapor, and energy flux densities," 2001.

[32] 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.

[33] http://diydrones.com.

[34] http://www.co2meter.com.