CSCI.6500/4500 Distributed Computing over the Internet—Programming Distributed Computing Systems (Varela)—Chapter 1

Instructor: Carlos Varela
Rensselaer Polytechnic Institute
Spring 2021
Overview of Distributed Computing over the Internet

• Goal 1: to understand and apply theoretical foundations of programming concurrent, distributed, and mobile computing systems

<table>
<thead>
<tr>
<th>Model</th>
<th>Language(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi$ calculus (Chapter 3)</td>
<td>Pict, Nomadic Pict (Chapter 8)</td>
</tr>
<tr>
<td>Actors (Chapter 4)</td>
<td>SALSA (Chapter 9)</td>
</tr>
<tr>
<td>Join Calculus (Chapter 5)</td>
<td>JoCaml (Chapter 10)</td>
</tr>
<tr>
<td>Mobile Ambients (Chapter 6)</td>
<td></td>
</tr>
</tbody>
</table>

• Goal 2: to compare communication and synchronization aspects in concurrent programming languages following these theoretical models.
Overview of Distributed Computing over the Internet

Goal 3: To review current research on distributed computing over the Internet

- coordination
- mobility
- heterogeneity
- security
- fault tolerance
- scalability
- programmability
- verification
Motivation

Digital computing only has a few decades of existence.

- Moore’s law predicted that the number of transistors per chip would double every two years, effectively going from 2,300 transistors in 1971 to over 2,600,000,000 in 2011.
- ARPANet’s 50,000 bits per second in the U.S. in 1968 to a U.S.-Japan backbone bandwidth of 500,000,000,000 bits per second in 2011.
- The first hard drive in 1956 capable of storing 4,400,000 bytes of data to relatively cheap personal hard drives capable of storing over 4,000,000,000,000 bytes of data in 2011

Six orders of magnitude improvement in four decades. \(2^{40/2} \approx 10^6\)
Motivation

By 2031, we may see a thousand-fold further improvement in hardware, resulting in

- over 1,000-core processors in mobile devices, laptops, and desktops,
- over 100 terabit per second network backbones, and
- hard drives storing over peta-bytes, exa-bytes.

The growing number of transistors in emerging multi-core architectures can only be translated into practical software applications’ performance if software developers understand how to program concurrent systems effectively.
Why Concurrency?

- *Concurrency* can be regarded as potential parallelism.
  - Computing activities can be executed in parallel provided there are no data or control dependencies that require them to be executed sequentially.
  - However, a concurrent computer program can also be executed on a single processor by distributing its activities over time, also known as *interleaving*.

- When multiple processors are used to execute a concurrent program, we say that it is a *parallel execution*.

- If the processors are separated geographically, we say that it is a *distributed execution*.

- If the location of the activities can change dynamically, we say it is a *mobile execution*. 
Concurrency Advantages

- Concurrent programs have the potential to execute much faster than their sequential counterparts, provided
  - independent sub-computations
  - enough resources
- If sub-computations are completely independent, these are called massively parallel applications.
- In the worst case scenario, sub-activities have significant data and control dependencies, and sequential execution represents the best execution time.
- Most applications lie in the middle. For example, a web server may be able to service client requests in parallel, but a database service component may have to serialize writing requests for consistency.
Concurrency Advantages

- Concurrent software enables *better human-computer interaction.*
  - For example, a mobile phone can be used for oral communication at the same time that its calendar or contacts features are used to find or update relevant information.

- Concurrent software also enables *better resource utilization.*
  - For example, a network interface card can send and receive data from other computers while the processor is busy computing application functions and the graphical co-processor is busy rendering images for data visualization.
Concurrency Challenges

Concurrency requires a different way of thinking about and organizing computer programs.

• As opposed to sequential programs, concurrent programs are typically not deterministic: different executions of the same program can produce different results.
  • The combinatorial explosion of possible execution orders makes reasoning about correctness and termination harder than it is for sequential code.
  • Programming abstractions designed to help tackle the inherent complexity of concurrent execution can be misused producing deadlocks.
Concurrency Challenges

- *safety*—that is, ensuring data consistency
- *liveness*—that is, ensuring computation progress
- *high performance* — additional resources required to schedule multiple activities and enable their communication and synchronization.
Why Distribution?

Distributed computing refers to computation that spreads over space and time.

- As an example, geographically separated database servers may each contain part of the *distributed state* of an application.

- Banking transactions started by the user of an automated teller machine in any part of the world, or even a simple Google search may trigger the response of dozens or even hundreds of servers worldwide.

Internet computing, web computing, grid computing, cloud computing, edge computing are all forms of *distributed computing*. 
Distribution Advantages

One significant advantage of distributed computation is the potential scalability afforded by applications.

- Data in many domains including science, engineering, finance, medicine, and others, is growing exponentially, and computing over the data is imperative to be able to analyze it and derive hypotheses, improve products, and generate wealth and well-being.

- As a consequence, the need to scale data analyses to large data sets requires ever larger computing infrastructures that can only be provided by multiple geographically distributed data and computing centers.
Distribution Challenges

Distributed systems are more difficult to reason about and to develop.

- Standards are often required to enable components developed by independent parties to interact.

- Failures can be partial, whereby only part of the application’s distributed state is available at a given point in time.

- The heterogeneity inherent in geographically distributed computing infrastructures makes resource scheduling and allocation critical to provide a reasonable quality of service to application users.

- The open nature of the Internet and the Web make security a key aspect of distributed systems. Data privacy and integrity are fundamental to many applications, for example, in the health and financial industries.
Why Mobility?

We use *mobility* to refer to mobile distributed systems which may include one or more of the following:

- mobile code
- mobile data
- mobile devices
- mobile users
Mobile Code

Mobile code can refer to

- downloading computer programs or components from the Internet, or

- applications whose components reconfigure themselves at run-time to adapt to changes in their execution environment.
Mobile Code Advantages

- Mobility of code enables dynamic application extensibility
  - web browsers automatically downloading applets or plug-ins, to augment their behavior or to enable viewing of heterogeneous data types.

- Mobility of code also enables distributed applications to adapt to changing network conditions.
  - a scientific data analysis application can grow to make use of thousands of newly available processors over the Internet by splitting and moving its components dynamically.
  - likewise, it can shrink by merging its components to run efficiently on a smaller number of resources as distributed computers become unavailable, fail, or are put to sleep to save energy.
Mobile Data

- *Mobile data* refers to accessing remote databases, for example as produced by sensors, and requires policies for replication, efficient access, and preservation.

- The mobility of data, including its potential replication, enables faster access by both human users and computer applications.
  - Data replication also enables fault tolerance.

- Sensor and actuator networks increasingly produce data at different scales continuously.

- Applications which produce or consume data continuously are known as *data streaming* applications.
Mobile Devices and Users

- *Mobile devices* refers to laptops, personal digital assistants, cellular phones, robots, cameras, cars, planes, and other devices that can change physical location and can become connected or disconnected at any time.

- Typically, mobile devices are highly heterogeneous, and can have significant constraints in terms of
  - processing speed,
  - network access (typically wireless,)
  - memory size, and
  - availability (e.g., limited life time batteries.)

- *Mobile users* need access to information and applications from anywhere anytime, even while traveling at different speeds and enjoying different levels of connectivity.
Mobility Advantages and Challenges

• Mobility enables
  • dynamically adaptable applications,
  • real-time streaming data analyses,
  • information access through heterogeneous devices and from disparate geographical locations.

• Mobility enables computing with sparse data sets in different application domains, including agriculture, space exploration, weather monitoring and forecasting, environmental sciences, etc.

• However, mobility also adds a new dimension of complexity to consider in distributed software development.
Models of Computation

- Theoretical models of computation are critical to be able to reason about properties of software, including the programming languages used to develop the software.

- Examples of software properties include
  - expressive power,
  - correctness,
  - performance,
  - reliability.

- Models of computation also help programming language designers and application developers organize software artifacts to handle complexity, and promote re-usability and composability.
Church and Kleene created the $\lambda$ calculus in the 1930’s.

- The calculus is at the heart of functional programming languages.
- It is Turing-complete, that is, any computable function can be expressed and evaluated using the calculus.
- It is useful to study sequential programming language concepts because of its high level of abstraction.
Π Calculus

Milner et al. (1992) created the π calculus

• it models concurrent computation in terms of communicating processes

• processes can dynamically change their communication topology

• Milner developed a theory of equivalence of processes
  • it uses a technique called bisimulation
    • two processes are considered equivalent if they can indefinitely mimic each other’s interactions with any environment.
Actors

Hewitt et al. created the *actor* model of concurrent computation in the 1970’s. Agha (1986) refined it to model open distributed systems.

- Actors model concurrent units of computation, which communicate by asynchronous and guaranteed message passing.

Agha et al. (1997) extended the call-by-value $\lambda$ calculus with primitives for actor creation and communication, and provided an operational semantics for the actor model.

- Sequential computation within an actor is modeled with the $\lambda$ calculus
- Actor configurations model communication and distributed state
  - evolve over time using a labelled transition system.
- A theory of observational equivalence was developed
Join Calculus

Fournet and Gonthier (1996) created the *join calculus* to model concurrent computation as chemical processes

- complex expressions can appear as the result of combining simpler expressions according to a dynamic set of bi-directional rewriting rules.

- The transformation of simpler expressions into more complex expressions and vice versa is inspired by heating molecules and cooling atoms in chemical reactions.

- An observational congruence is defined as a behavioral *may* equivalence for a restricted yet equally expressive subset (i.e., a core) of the join calculus.
Mobile Ambients

Cardelli and Gordon (2000) created the theory of mobile ambients to model mobile code, data, devices, and users.

- The ambient calculus defines a hierarchical structure of ambients which contain processes and other ambients.
- Processes are restricted to only communicate within the same ambient.
- Primitives in the calculus enable the movement of ambients in and out of other ambients, as well as creation and destruction of ambient boundaries.
- Ambient calculus expressions can be translated into the pure $\pi$ calculus, and vice versa.
  - This strategy enables the use of equivalence techniques developed for the $\pi$ calculus in the ambient calculus.
Programming Languages

We can categorize programming languages according to their programming models or paradigms


• *functional*—Lisp (1958), ML (1973), Scheme (1975), Haskell (1987)

• *logical*—Prolog (1972), Mercury (1995)


All paradigms have been used whether with libraries or with language extensions for concurrent programming.
Concurrent programming with processes

- Processes have been a key abstraction used by operating systems to isolate applications from one another when running on the same computer.

- Processes have also been widely used by parallel application developers, e.g., using the Message Passing Interface (MPI) as an abstraction to separate parallel computation running on multiple computers.

- We study a programming language that follows the $\pi$ calculus model and adds the notion of types to inter-process communication channels: Pict (Pierce and Turner, 2000).

- We also study an extension of Pict to study process mobility: Nomadic Pict (Wojciechowski and Sewell, 1998).
Concurrent programming with actors

Actor oriented programming languages include:

- PLASMA (Hewitt 1975)
- Act (Lieberman 1981)
- ABCL (Yonezawa et al., 1986)
- Actalk (Briot 1989)
- Erlang (Armstrong et al., 1993)
- SALSA (Varela and Agha, 2001)
- E (Miller et al., 2005)
- AmbientTalk (Dedecker et al., 2006)

There are also many actor libraries and frameworks, e.g., Akka for Scala (Bonér 2009).
Concurrent programming with actors

In response to a message, an actor can:

1. modify its local state,
2. create new actors, and/or
3. send messages to acquaintances.
Concurrent programming with join patterns

• A programming language based on the join calculus is JoCaml, which extends OCaml with join patterns (Fournet et al., 2002).

• OCaml is an object-oriented extension of Caml.

• Caml is an efficient implementation of ML.

• C# has borrowed some notions from join patterns to implement synchronization at a higher level (Benton et al., 2002).
Programming Distributed Computing Systems

In this course, we will use Pict, Nomadic Pict, SALSA and JoCaml to illustrate practical aspects of developing concurrent, distributed, and mobile systems.

<table>
<thead>
<tr>
<th>Model</th>
<th>Language(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi$ calculus (Chapter 3)</td>
<td>Pict, Nomadic Pict (Chapter 8)</td>
</tr>
<tr>
<td>Actors (Chapter 4)</td>
<td>SALSA (Chapter 9)</td>
</tr>
<tr>
<td>Join Calculus (Chapter 5)</td>
<td>JoCaml (Chapter 10)</td>
</tr>
<tr>
<td>Mobile Ambients (Chapter 6)</td>
<td></td>
</tr>
</tbody>
</table>

These languages follow respectively the $\pi$ calculus, the actor model, and the join calculus theories.
Common Concurrent Programming Examples

For each theory of concurrent computation and for each concurrent programming language, we will illustrate how to encode the same three programs:

- Reference Cell
- Mutual Exclusion
- Dining Philosophers
Reference Cell

- A reference cell models stateful computation, i.e., a cell contains mutable state.

- A simple cell can be created, updated, and queried by other concurrent computations.

- A reference cell is perhaps the simplest stateful concurrent program that illustrates issues such as non-determinism:
  - e.g., if both an update and a query are concurrently issued, the order of request processing will determine the result received by the querying process.

- Even though a reference cell is a very simple example, it is at the basis of all stateful computation, including databases and more complex information systems.
Mutual Exclusion

*Mutual exclusion* refers to the need of two or more concurrent computations to coordinate and avoid executing at the same time some critical region of code.

- For example, if two computations are using a shared resource—say, a printer—it is important that they take turns, or else the result will not be as desired: e.g., you may end up with words from two different documents printed on the same page.

It illustrates the need for coordination of concurrent activities.

- A useful abstraction for guaranteeing mutual exclusion is the notion of *semaphores*.
- It forms the basis for more complex coordination mechanisms such as distributed transactions.
Dining Philosophers

A typical example of the complexities of concurrent computation is the famous *dining philosophers* scenario:

- Consider $n$ philosophers, $Ph_0, \ldots, Ph_{n-1}$, dining in a round table containing $n$ chopsticks $ch_0, \ldots, ch_{n-1}$ where each chopstick is shared by two consecutive philosophers.

Each philosopher uses the following sequential algorithm:

1. Pick up the *left* chopstick, if available.
2. Pick up the *right* chopstick, if available.
3. Eat.
4. Release both chopsticks.
5. Think.
6. Go to 1.
Dining Philosophers

While each philosopher’s algorithm seems to make sense from an individual sequential perspective, the group’s concurrent behavior has the potential for *deadlock*:

- all philosophers can pick up their left chopsticks in Step 1, and
- block forever in Step 2, waiting for the philosopher on her/his right to release her/his left chopstick.

This well-known example highlights the potential for *deadlock* in concurrent systems with shared resources.