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 (Part I)</th>
<th>Language(s) (Part II)</th>
</tr>
</thead>
<tbody>
<tr>
<td>π 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
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 Part II, we use different well-founded languages to illustrate practical aspects of developing concurrent, distributed, and mobile systems:

- Pict, Nomadic Pict (Chapter 8)
- SALSA (Chapter 9)
- JoCaml (Chapter 10)

The objective is two-fold:

- first, to bridge the gap between the theory and the practice of programming concurrent systems; and
- second, to illustrate the challenges that real-world distributed computing imposes on going from purely theoretical models to practical programming languages and systems.
Common Distributed Programming Examples

For each programming language, we will illustrate how to encode the same programs:

- Reference Cell
  - A Mobile 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.
A Mobile Reference Cell

A mobile reference cell is an archetypal distributed and mobile computing example:

- It illustrates *strong mobility*, i.e., mobile stateful computation.
- This is in contrast to *weak mobility*, where code can move from a computer to another, but it needs to start from a new state after movement.

Even though a mobile reference cell is a very simple example, it is at the basis of all distributed data structures (e.g., distributed lists, trees, graphs).
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.
Other Approaches to Concurrency

Consider the Java code (from Varela and Agha, 1998):

class A implements Runnable{
    B b;
    synchronized void m()
        {...b.n();...}
    public void run() { m(); }
}

class B implements Runnable{
    A a;
    synchronized void n()
        {...a.m();...}
    public void run() { n(); }
}

class Deadlock {
    public static void main(String[] args){
        A a = new A();   B b = new B();
        a.b = b;   b.a = a;
        Thread t1 = new Thread(a).start();
        Thread t2 = new Thread(b).start();
    }
}
Concurrent Object-Oriented Programming

The Java example illustrates a problem that results from having two different programming language abstractions for handling state and for handling concurrency, namely *objects* and *threads*:

- an object abstracts over a unit of state management, whereas
- a thread abstracts over a unit of concurrency.
Concurrent Object-Oriented Programming

The best that an object of class A can do is to use the synchronized keyword to prevent multiple threads from entering its m method simultaneously, requiring them to first obtain a re-entrant lock associated to the object. The same is true for class B and its n method.

- In this example, however, a possible race condition arises if thread t1 enters object a’s m method (obtaining a’s lock in the process) while thread t2 enters object b’s n method (also obtaining b’s lock in the process) and then both threads hang:
  - t1 waiting for t2 to release b’s lock in order to enter b’s n method and
  - thread t2 waiting for t1 to release a’s lock in order to enter a’s m method.
Concurrent Object-Oriented Programming

Since not using the `synchronized` keyword would lead to safety problems—i.e., inconsistent views of (or modifications to) an object’s state,—these *race conditions*, often leading to *deadlocks*, are a very common problem with concurrent programs written in concurrent object-oriented programming languages.
Other Approaches to Concurrency

Consider another example in the Oz programming language (Roy and Haridi, 2004):

\[
\begin{align*}
\text{local } & X \ Y \text{ in} \\
& \{\text{Browse } X\} \\
& \text{thread } \{\text{Delay 1000} \} \ Y = 10 \text{ end} \\
& X = Y + 100 \\
\end{align*}
\]

- This example illustrates two dataflow variables, namely \( X \) and \( Y \).
- The value of \( X \) depends on the value of \( Y \), which is to be assigned to in a separate child thread, which in this example suspends for a second to simulate an expensive computation.
- The main (parent) thread may therefore reach the expression requiring a value for \( Y \) before a value has been produced.
Dataflow Concurrency

Different programming languages behave differently in this scenario:

- The C programming language with threads, for example, would use whatever value appears in Y’s memory location at the time of reaching the expression needing its value.

- The Java programming language would assign a default value to the Y variable according to its type at initialization time, e.g., 0 for integers, which would be used by the thread requiring Y’s value.

- The Prolog programming language would raise an error dynamically stating that the value of Y has not been initialized—a natural behavior in a non-concurrent programming language.
Deterministic Concurrency

The Oz programming language, on the other hand, has the behavior to \textit{suspend} the computation of $X$ until such time as the value for $Y$ is produced by another thread.

In this particular example, Oz’s behavior gives us \textit{deterministic concurrency}:

- Independently of the thread scheduling of the parent and child threads, the value of the dataflow variable $X$ will always be the same: 110.
Deterministic Concurrency

Unfortunately, this deterministic concurrent behavior only arises if there is exactly one binder for each single-assignment dataflow variable. For example, these two snippets of Oz code:

```oz
local X Y in
    {Browse X}
    thread Y = 10 end
    thread Y = 20 end
    X = Y + 100
dend
```

```oz
local X Y in
    {Browse X}
    thread if false then Y = 10 end end
    X = Y + 100
dend
```

- will non-deterministically assign either 10 or 20 to Y and will cause an incompatible assignment exception to be thrown in the other thread (left), and

- will silently hang waiting for another thread to assign a value to Y, which will never occur (right).
Languages for Distributed Computing

To help bridge the gap between theory and practice, we will consider programming languages that closely follow the concurrency models studied in the first part.

- Pict is a programming language that uses the $\pi$ calculus as its foundation.
  - Nomadic Pict is an extension to the Pict language to support distribution and mobility aspects.
- SALSA is a programming language that implements the actor model of computation, including support for distribution and mobility.
- JoCaml is a programming language implementing the join calculus, as well as the distributed join calculus, an extension with distribution and mobility primitives.
Languages for Distributed Computing

In contrast to concurrency models that can afford very high levels of abstraction, programming languages have to specify “real-world” system issues, such as

- interaction with humans or other programs via standard input and output interfaces,
- a richer type system with complex values being passed in messages, and
- message patterns for easier expression of varied behaviors.
Languages for Distributed Computing

Distribution and mobility issues, which are sometimes abstracted by concurrency models and “left to the infrastructure”, need to also be specified as formally as possible.

• As a consequence, a programming language’s operational semantics is often more complex than the operational semantics of its corresponding model introduced in Part I.

• We strive for conciseness, and focus on the semantic aspects critical to concurrency, distribution, and mobility.
Distribution and Mobility Aspects—Pict

Pict was designed as an attempt to “program with the \( \pi \) calculus,” yet significant higher-level abstractions were built to make it look and feel like a functional programming language.

- The *synchronous* communication primitive of the \( \pi \) calculus was eliminated: Pict only supports asynchronous output.
- Sources of non-determinism were eliminated when possible:
  - e.g., \( P + Q \), non-deterministic choice, is not directly supported.
  - Also, replication was restricted to replicated input and *recursive definitions* were introduced to ensure that some channels have only one receiver process, effectively disallowing expressions such as \( a(x).P | a(x).Q | \bar{a}y \) which lead to non-deterministic execution of \( P \) or \( Q \).
Distribution and Mobility Aspects—Pict

Pict does not support distribution and mobility of processes. A language extension, Nomadic Pict, extends Pict with the notion of sites and agents.

- Communication channels belong to agents, which can migrate between sites.
- Nomadic Pict has constructs for location-independent communication, which can be specified atop lower-level location-dependent communication primitives.
- The distinct abstraction for mobile processes (i.e., named agents) means there are two kinds of processes: stationary and mobile.
  - channels can refer to a stationary process or to a mobile agent, creating a dichotomy which confuses mobility semantics.
Distribution and Mobility Aspects—SALSA

SALSA is an extension of Java to support the actor model of concurrent and distributed computation.

- SALSA adds the notion of *universal actors* which have unique names and enable transparent actor migration and location-independent communication using Internet protocols.
- A *naming service* ensures uniqueness of universal actor names using the Domain Name Service (DNS).
- A *theater* run-time system is a Java VM with support for
  - remote actor creation,
  - actor migration,
  - remote message sending, and
  - access to local *environment* actors.
Distribution and Mobility Aspects–SALSA

Actor migration is easy to implement: in response to a `migrate` message, SALSA

- serializes the actor’s internal state
- serializes the actor’s mailbox of pending messages, and
- restarts the actor’s execution in the new theater.
- The naming service is then informed of the actor’s new location.
  - A temporary proxy ensures messages are delivered to the new location.

SALSA supports automatic garbage collection of local, distributed, and mobile actors.

Other actor programming languages (e.g., Erlang and Scala) do not support actor mobility and actor garbage collection.
Distribution and Mobility Aspects—JoCaml

JoCaml is an extension of Objective Caml to support the join calculus and the distributed join calculus, and extension of the join calculus with additional primitives for distribution, mobility, and failure detection.

- Communication is location-independent.
- The JoCaml run-time system adds the notions of a name server and sites to support distribution.
- Locations are organized in JoCaml hierarchically and move with all its contained locations and processes, using subjective mobility, which resembles mobile ambients.
- Locations can fail (failing all its nested locations) and a failure detection primitive, in restricted cases, allows to detect when a location or any of its parent locations has failed.