Concurrent and Distributed Patterns

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

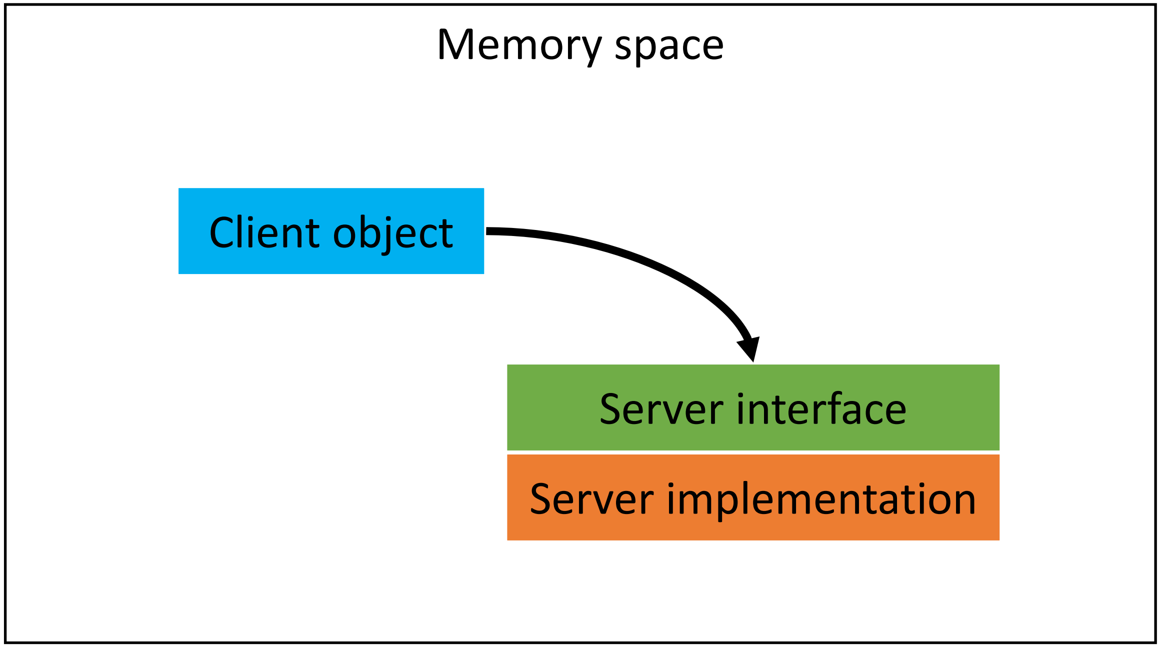
Outline

- Distributed OO
- Concurrency problems
- Possible solutions:
  - Synchronization patterns
  - Context patterns
  - Request/event handling patterns

Distributed OO

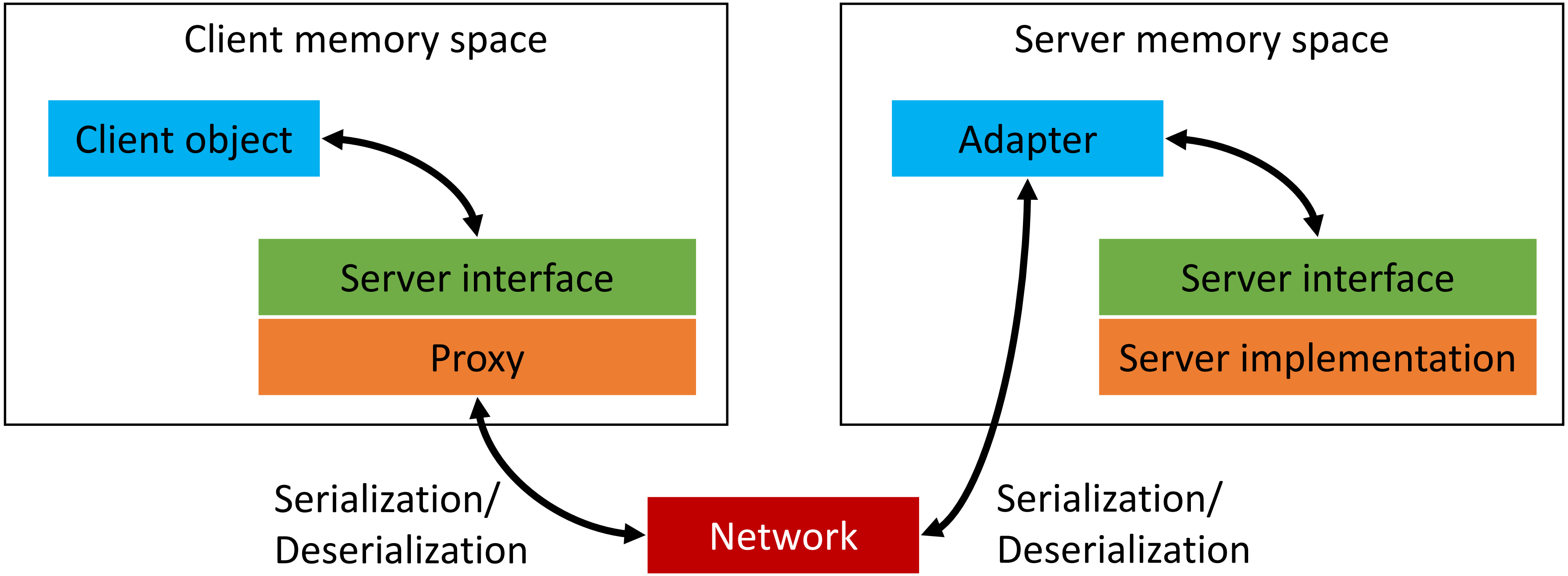
Local call

- Caller object: client
- Called object: server
- They are in the same memory space:



Remote call

- Proxy (stub): appears as if the server was local
  - connects to server, serializes parameters, deserializes result
- Adapter: publishes the server implementation on the network
  - accepts client requests, deserializes parameters, calls server implementation, serializes result



Problems introduced by remote communication

- Heterogeneity: different programming languages
- Memory management problems
  - separate memory spaces for the client and the server
  - parameters and results have to be serialized and deserialized
  - pointers/references have to be serialized recursively
  - memory allocation and freeing
  - preserving server state between calls
- Network problems
  - problem if the server is unavailable
  - problem if the client is unavailable
  - data integrity
- Concurrency problems in the server
  - multiple clients
  - multiple threads
- Latency problems
  - large response times
  - long running operations
  - synchronous and asynchronous calls
- Hard to monitor, hard to debug

Concurrency problems

Concurrency problems

- Mutable shared state
- Race conditions
- Synchronization
- Dead-locks
- Starvation
- Can’t be exactly reproduced
- Can’t be tested
- High complexity

Unpredicted thread shcedule

Synchronization patterns

Synchronization patterns

- Synchronizing the execution of threads
- Patterns:
  - Critical sections: operations appear atomic
    - Atomic operations
    - Scoped locking
  - Balking: wait until the object is in the appropriate state
    - Balking design pattern
    - Double-checked locking
    - Guarded suspension
  - Signaling: notifying other threads
    - Monitor object
    - Mutex
    - Semaphore
    - AutoResetEvent
    - ManualResetEvent
    - Readers-writer lock

Synchronization patterns

Critical sections: operations appear atomic

Atomic operation

- Appears to the rest of the system to occur instantaneously
- Guarantee of isolation from concurrent processes
- C#: System.Threading namespace

```
// ++counter;
Interlocked.Increment(ref counter);
// tmp = obj; obj = value; return tmp;
tmp = Interlocked.Exchange(ref obj, value);
// tmp = obj; if (obj == coparand) { obj = value; } return tmp;
tmp = Interlocked.CompareExchange(ref obj, value, comparand);
```

- Java: java.util.concurrent.atomic package

```
// ++counter;
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet();
// tmp = obj; obj = value; return tmp;
AtomicReference<Object> obj = new AtomicReference<Object>();
tmp = obj.set(value);
// if (obj == coparand) { obj = value; return true; } else { return false; }
obj.compareAndSet(comparand, value);
```

Scoped locking

- Defining critical sections
- Grouping multiple operations as if they were atomic
- Scoped locks in .NET and Java are reentrant: recursive calls on the same thread do not cause dead-lock

Keeps track how many times it enteredk

- C#: lock keyword

```
lock (obj)
{
    // ...
}
```

- Java: synchronized keyword

```
synchronized (obj) {
    // ...
}
```

Synchronization patterns

Balking: wait until the object is in the appropriate state

Balking

- If a method is invoked when the object is in an inappropriate state, then the method will return without doing anything or wait until the state is appropriate
- Balking patterns:
  - Balking design pattern: perform an operation only in a given state, otherwise return without doing anything
    - e.g. the operation is already in progress
  - Double-checked locking optimization: acquire a lock only if necessary
    - e.g. lazy initialization of a singleton object
  - Guarded suspension: wait until a lock is acquired and a pre-condition is met
    - e.g. wait until there is something to process

Balking design pattern

- Return immediately if the state is inappropriate:
  - e.g. job is already in progress

```
public class Example {
    private bool jobInProgress = false;

    public void ExecuteJob() {
        lock (this) {
            if (jobInProgress) {
                return;
            }
            jobInProgress = true;
        }

        // Code to execute job goes here
        // ...

        lock (this) {
            jobInProgress = false;
        }
    }
}
```

Double-checked locking

- Problem of implementing a singleton:

```
public class Singleton
{
    private static Singleton singleton = null;

    private Singleton() { }

    public static Singleton GetInstance()
    {
        if (singleton == null)
        {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

Problem:
two threads may enter the “if” simultaneously:
singleton is created twice

Double-checked locking

- A possible solution:

```
public class Singleton
{
    private static object myLock = new object();
    private static Singleton singleton = null;

    private Singleton() { }

    public static Singleton GetInstance()
    {
        lock (myLock)
        {
            if (singleton == null)
            {
                singleton = new Singleton();
            }
            return singleton;
        }
    }
}
```

Problem:
lock/synchronized is expensive,
and it is executed every time the method is called

Double-checked locking

- Double-checked locking (OK in .NET 2.0+, but not in .NET 1 or Java):

```
public class Singleton
{
    private static object myLock = new object();
    private static Singleton singleton = null;

    private Singleton() {}

    public static Singleton GetInstance()
    {
        if (singleton == null)
        { // 1st check
            lock (myLock)
            {
                if (singleton == null)
                { // 2nd (double) check
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

<u>More efficient:</u>
lock/synchronized is called only if necessary

<u>Problem in Java and .NET 1:</u>
pointer is set before the constructor is finished executing, the second thread can return a partially constructed object

Double-checked locking

- Solution (OK in .NET 1+ and Java 5+, but not Java 4-): volatile

```
public class Singleton
{
    private static object myLock = new object();
    private volatile static Singleton singleton = null;

    private Singleton() {}

    public static Singleton GetInstance()
    {
        if (singleton == null)
        { // 1st check
            lock (myLock)
            {
                if (singleton == null)
                { // 2nd (double) check
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

Problem in Java 4-:
same as before, since the semantics of volatile was only corrected in Java 5

Double-checked locking

- Very hard to implement it correctly
  - due to compiler optimizations
  - due to processor optimizations
- .NET 2.0+:
  - works without volatile, since the lock keyword enforces correct execution order
- .NET 1, Java 5+:
  - works only with volatile
- Java 4-:
  - There is no correct solution for the double-checked locking, don't use it!

Avoiding double-checked locking

- Using static initialization (always correct in .NET and Java):

```
public class Singleton
{
    private static readonly Singleton singleton = new Singleton();

    private Singleton() {}

    public static Singleton GetInstance()
    {
        return singleton;
    }
}
```

Problems:
- it is not lazily initialized
- static initialization order is not deterministic
- cannot be used for non-static (non-singleton) cases

Avoiding double-checked locking

- Using static lazy initialization (always correct in .NET and Java):

```
public class Singleton
{
    // Lazy initialization:
    private class Holder
    {
        public static readonly Singleton singleton = new Singleton();
    }

    private Singleton() {}

    public static Singleton GetInstance()
    {
        return Holder.singleton;
    }
}
```

should be a static class in Java

Problem:
- cannot be used for non-static (non-singleton) cases

Guarded suspension

- Waiting until a lock is acquired and a precondition is met:

```
public void operationWithPrecondition() {
  synchronized (lock) {
    while (!preCondition) {
      try {
        lock.wait();
      } catch (InterruptedException e) { }
    }
    // ...
  }
}

public void fulfillPrecondition() {
  synchronized (lock) {
    preCondition = true;
    lock.notify();
  }
}
```

Must be a while, not an if:
when the thread is awoken or interrupted,
the pre-condition may still not be met

The wait() exits from the synchronized block:
other threads can enter the synchronized block

(Never create an empty while loop!
It is busy waiting and consumes very high CPU!)

It is not deterministic which
thread is awoken

Guarded suspension

- Example: FIFO

```
public class Fifo<T> {
  private Object lock = new Object();
  private ArrayList<T> items =
                     new ArrayList<>();

  public void enqueue(T item) {
    synchronized (lock) {
      while (items.size() > 10) {
        try {
          lock.wait();
        }
        catch (InterruptedException e)
        {}
      }
      items.add(item);
      lock.notifyAll();
    }
  }
```

```
  public T dequeue() {
    T result;
    synchronized (lock) {
      while (items.size() == 0) {
        try {
          lock.wait();
        }
        catch (InterruptedException e)
        {}
      }
      result = items.get(0);
      items.remove(0);
      lock.notifyAll();
    }
    return result;
  }

  // ...
}
```

fulfill pre-condition

Synchronization patterns

Signaling: notifying other threads

Signaling

- Threads notify other threads
  - e.g. in the case of guarded suspension
- Cases:
  - Monitor object: mutual exclusion + signaling other threads that their waiting condition has been met
  - Mutex: only one thread can enter (mutual exclusion)
  - Semaphore: only a given number of threads can enter
  - ManualResetEvent: allow multiple threads to continue after an operation is done
  - AutoResetEvent: allow a single thread to continue after an operation is done
  - Readers-writer lock: allow multiple reads but only a single write
- C#: common base class for signals is System.Threading.WaitHandle
- Java: no common base class for signals

Monitor object

- Allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true
- Provides a mechanism for signaling other threads that their condition has been met
- Java: every object is a Monitor object
  - mutual exclusion: synchronized keyword with the object as parameter
  - signaling: wait(), notify(), notifyAll()
- C#: System.Threading.Monitor static class
  - provides static utility functions for treating regular objects as Monitor objects
  - mutual exclusion: Enter(object), Exit(object)
  - signaling: Wait(object), Pulse(object), PulseAll(object)

Monitor object: Java

- Example: FIFO (same as the guarded suspension example)

in this example this is the monitor object

```
public class Fifo<T> {
  private Object lock = new Object();
  private ArrayList<T> items =
                      new ArrayList<>();

  public void enqueue(T item) {
    synchronized (lock) {
      while (items.size() > 10) {
        try {
          lock.wait();
        }
        catch (InterruptedException e)
        {}
      }
      items.add(item);
      lock.notifyAll();
    }
  }

  public T dequeue() {
    T result;
    synchronized (lock) {
      while (items.size() == 0) {
        try {
          lock.wait();
        }
        catch (InterruptedException e)
        {}
      }
      result = items.get(0);
      items.remove(0);
      lock.notifyAll();
    }
    return result;
  }

  // ...
}
```

fulfill pre-condition

Monitor object: C#

- Example: FIFO (same as the guarded suspension example)

in this example this is the monitor object

```
public class Fifo<T>
{
    private object obj = new object();
    private List<T> items = new List<T>();

    public void Enqueue(T item)
    {
        Monitor.Enter(obj);
        try
        {
            while (items.Count > 10)
            {
                Monitor.Wait(obj);
            }
            items.Add(item);
            Monitor.PulseAll(obj);
        }
        finally
        {
            Monitor.Exit(obj);
        }
    }
```

fulfill pre-condition

```
    public T Dequeue()
    {
        Monitor.Enter(obj);
        try
        {
            T result;
            while (items.Count == 0)
            {
                Monitor.Wait(obj);
            }
            result = items[0];
            items.RemoveAt(0);
            Monitor.PulseAll(obj);
            return result;
        }
        finally
        {
            Monitor.Exit(obj);
        }
    }
}
```

Semaphore

- Used to control access to a pool of resources
  - we have k resources and k keys
  - anyone who has a key can access a resource
  - example: k toilets with identical locks and keys
- A semaphore is decremented each time a thread enters the semaphore
- A semaphore is incremented when a thread releases the semaphore
- A semaphore blocks if the counter reaches zero and the thread tries to enter
- C#: System.Threading.Semaphore

```
Semaphore semaphore = new Semaphore(initialCount, maximumCount);
// Decrease counter:
semaphore.WaitOne();
// Increase counter:
semaphore.Release();
```

- Java: java.util.concurrent.Semaphore

```
Semaphore semaphore = new Semaphore(MAX_COUNT);
// Decrease counter:
semaphore.acquire();
// Increase counter:
semaphore.release();
```

Mutex

- Synchronization primitive that grants exclusive access to the shared resource to only one thread
- If a thread acquires a mutex, the second thread that wants to acquire that mutex is suspended until the first thread releases the mutex
- Example: a mutex is a key to a toilet, one person can occupy the toilet at a time
- Mutex is the same as Semaphore with counter = 1
- C#: System.Threading.Mutex class

```
Mutex mutex = new Mutex();
// Acquire:
mutex.WaitOne();
// Release:
mutex.ReleaseMutex();
```

- Java: java.util.concurrent.Semaphore with max. count = 1

Manual reset event

- Allow multiple threads to continue after an operation is done
- It is like a door, which needs to be closed (reset) manually
  - people can go through as long as the door is open
- Two states:
  - signaled (set): threads are allowed to continue
  - non-signaled (reset): threads are blocked
- C#: System.Threading.ManualResetEvent
  - Set(): make it signaled
  - Reset(): make it non-signaled
  - WaitOne():
    - returns immediately and allows the thread to continue if the event is signaled
    - blocks if the event is non-signaled
- Java: no such solution, but can be implemented easily

Implementation of manual reset event

```java
public class ManualResetEvent {

  private final Object monitor = new Object();
  private volatile boolean signaled = false;

  public ManualResetEvent(boolean signaled) {
    this.signaled = signaled;
  }

  public void set() {
    synchronized (monitor) {
      signaled = true;
      monitor.notifyAll();
    }
  }

  public void reset() {
    synchronized (monitor) { // required only in Java 4-
      signaled = false;
    }
  }
```

Implementation of manual reset event

```java
public void waitOne() {
  synchronized (monitor) {
    while (!signaled) {
      try {
        monitor.wait();
      } catch (InterruptedException e) {
        // nop
      }
    }
  }
}
```

Implementation of manual reset event

```java
public boolean waitOne(long timeout) {
  synchronized (monitor) {
    long t = System.currentTimeMillis();
    while (!signaled) {
      try {
        monitor.wait(timeout);
      } catch (InterruptedException e) {
        // nop
      }
      // Check for timeout
      if (System.currentTimeMillis() - t >= timeout) {
        break;
      }
    }
    return signaled;
  }
}
```

Manual reset event example

```csharp
// Thread 1:
public class Downloader
{
    public ManualResetEvent Downloaded { get; }

    public Downloader()
    {
        this.Downloaded = new ManualResetEvent(false);
    }

    public void Download()
    {
        this.Downloaded.Reset();
        // ... looong operation ...
        this.Downloaded.Set();
    }
}
```

```csharp
// Thread 2:
public class FileOpener
{
    private Downloader downloader;

    public FileOpener(Downloader downloader)
    {
        this.downloader = downloader;
    }

    public void OpenFile()
    {
        this.downloader.Downloaded.WaitOne();
        // ... open downloaded file ...
    }
}
```

Auto reset event

- Allow a single thread to continue after an operation is done
- It is like a tollbooth
  - allows one car to go by and automatically closes before the next one can get through
- Same as a Manual reset event, except that WaitOne() automatically calls Reset(), so other threads won't be able to continue
- C#: System.Threading.ManualResetEvent
  - Set(): make it signaled
  - Reset(): make it non-signaled
  - WaitOne():
    - returns immediately and allows the thread to continue if the event is signaled
    - blocks if the event is non-signaled
    - calls Reset() before returning, so other threads will be blocked
- Java: no such solution, but can be implemented easily

Implementation of auto reset event

```java
public class AutoResetEvent {

  private final Object monitor = new Object();
  private volatile boolean signaled = false;

  public AutoResetEvent(boolean signaled) {
    this.signaled = signaled;
  }

  public void set() {
    synchronized (monitor) {
      signaled = true;
      monitor.notifyAll();
    }
  }

  public void reset() {
    synchronized (monitor) { // required only in Java 4-
      signaled = false;
    }
  }
```

Implementation of auto reset event

```
public void waitOne() {
  synchronized (monitor) {
    while (!signaled) {
      try {
        monitor.wait();
      } catch (InterruptedException e) {
        // nop
      }
    }
    signaled = false;
  }
}
```

Implementation of auto reset event

```java
public boolean waitOne(long timeout) {
  synchronized (monitor) {
    try {
      long t = System.currentTimeMillis();
      while (!signaled) {
        try {
          monitor.wait(timeout);
        } catch (InterruptedException e) {
          // nop
        }
        // Check for timeout
        if (System.currentTimeMillis() - t >= timeout) {
          break;
        }
      }
      return signaled;
    } finally {
      signaled = false;
    }
  }
}
```

Auto reset event example

```
public class PrinterSpooler
{
    private AutoResetEvent printerGuard;

    public PrinterSpooler()
    {
        printerGuard = new AutoResetEvent(true);
    }

    public void Print(PrintJob printJob)
    {
        this.printerGuard.WaitOne();
        // If we reach here, we have sole access to the printer.
        // ... print the job
        this.printerGuard.Set();
    }
}
```

ManualResetEvent and AutoResetEvent

- Warning! Don’t do this:

```
public class Periodic
{
    public AutoResetEvent Guard { get; }

    public void PeriodicSignal()
    {
        while (true)
        {
            this.Guard.Set(); // Wrong! May not signal the other thread!
            this.Guard.Reset(); // Because Reset() may run too soon!
            Thread.Sleep(1000);
        }
    }
}
```

- For generating a periodic signal use Monitor instead:
  - notify() or Pulse() inside the loop (instead of Set()-Reset())
  - wait() or Wait() from other threads (instead of WaitOne())

Readers-writer lock

- Accessing a single resource
- Readers can run in parallel as long as the resource is not written
- Writers get exclusive access, only a single writer can run at a time, and also readers are blocked until the writer is finished
- When the writer tries to acquire a lock:
  - existing readers will be allowed to finish
  - the writer is blocked until then
  - new readers will be blocked
  - when all existing readers are finished, the writer gets the lock
  - when the writer is finished, readers can run again
- Be careful with recursion! It may cause dead-locks!
- C#: System.Threading.ReaderWriterLockSlim
- Java: java.util.concurrent.locks.ReentrantReadWriteLock

Context patterns

Context patterns

- Provide context specific information for threads
- Patterns:
  - Global context
  - Thread-local storage
  - Thread-local context

Global context

- Provides a globally accessible context (execution environment)
- Useful if there is no dependency injection and we do not want to pass the context to every method
- It is like a static singleton, but available only in a given scope
  - careful: other threads may run outside the scope! In a multi-threaded application use Thread-local context!
- Example:

```
public class SomeClass {
    public void SomeMethod() {
        // Access the value stored in GlobalContext: "Hello"
        string currentValue = GlobalContext.Current.SomeValue;
    }
}
public class SomeProgram {
    public static void Main(string[] args) {
        // Make GlobalContext available only in this scope:
        using (var scope = new GlobalContextScope("Hello")) {
            SomeClass cls = new SomeClass();
            cls.SomeMethod();
        }
    }
}
```

Implementation of GlobalContext

```
public class GlobalContext
{
    // Instantiated only by GlobalContextScope:
    internal GlobalContext(string value)
    {
        this.SomeValue = value;
    }

    // An option/value available in the context:
    public string SomeValue { get; }

    // Gets the current context, set only by GlobalContextScope:
    public static GlobalContext Current { get; internal set; }
}
```

Implementation of GlobalContextScope

```
public class GlobalContextScope : IDisposable
{
    private static object lockObj = new object();

    // Set the GlobalContext if it is not yet set:
    public GlobalContextScope(string value) {
        lock (lockObj) {
            if (GlobalContext.Current != null) {
                throw new InvalidOperationException(
                    "The global context is already set.");
            }
            GlobalContext.Current = new GlobalContext(value);
        }
    }

    // Remove the GlobalContext if the scope is ended:
    public void Dispose() {
        lock(lockObj) {
            GlobalContext.Current = null;
        }
    }
}
```

Thread-local storage

- Static fields in .NET and Java are specific to a class across all objects and all threads
- Thread-local storage is usually a static field that stores thread-specific value
  - same value across all objects
  - but a different value for each thread
  - acts like a Dictionary/HashMap where the key is the thread
- Thread-local instance (i.e. non-static) fields are rarely used
- C#: System.Threading.ThreadLocal<T>
- Java: java.lang.ThreadLocal<T>

Thread-local context

- Provides a globally accessible thread-specific context (execution environment)
- Useful if there is no dependency injection and we do not want to pass the context to every method
- Similar to a Global context, but implemented with thread-local storage
- Can be used on the server side to provide execution context for worker threads
  - e.g. WCF OperationContext

Thread-local context example

```
public class SomeClass {
    public void SomeMethod() {
        // Access the value stored in ThreadLocalContext: "Hello"
        string currentValue = ThreadLocalContext.Current.SomeValue;
    }
}
public class SomeThread {
    public void Run() {
        // Make ThreadLocalContext available only in this scope:
        using (var scope = new ThreadLocalContextScope("Hello")) {
            SomeClass cls = new SomeClass();
            cls.SomeMethod();
        }
    }
}
```

Implementation of thread-local context

```
public class ThreadLocalContext
{
    // Instantiated only by ThreadLocalContextScope:
    internal ThreadLocalContext(string value)
    {
        this.SomeValue = value;
    }

    // An option/value available in the context:
    public string SomeValue { get; }

    private static ThreadLocal<ThreadLocalContext> current =
        new ThreadLocal<ThreadLocalContext>();

    // Gets the current context, set only by ThreadLocalContextScope:
    public static ThreadLocalContext Current
    {
        get { return ThreadLocalContext.current.Value; }
        internal set { ThreadLocalContext.current.Value = value; }
    }
}
```

Implementation of thread-local context scope

```
// Create a separate scope for each thread.
// Locking is not necessary any more, since only the current thread
// has access, and hence there is no shared state between threads:
public class ThreadLocalContextScope : IDisposable
{
    // Set the ThreadLocalContext if it is not yet set:
    public ThreadLocalContextScope(string value)
    {
        if (ThreadLocalContext.Current != null)
        {
            throw new InvalidOperationException(
                "The thread-local context is already set.");
        }
        ThreadLocalContext.Current = new ThreadLocalContext(value);
    }

    // Remove the ThreadLocalContext if the scope is ended:
    public void Dispose()
    {
        ThreadLocalContext.Current = null;
    }
}
```

Request/event handling patterns

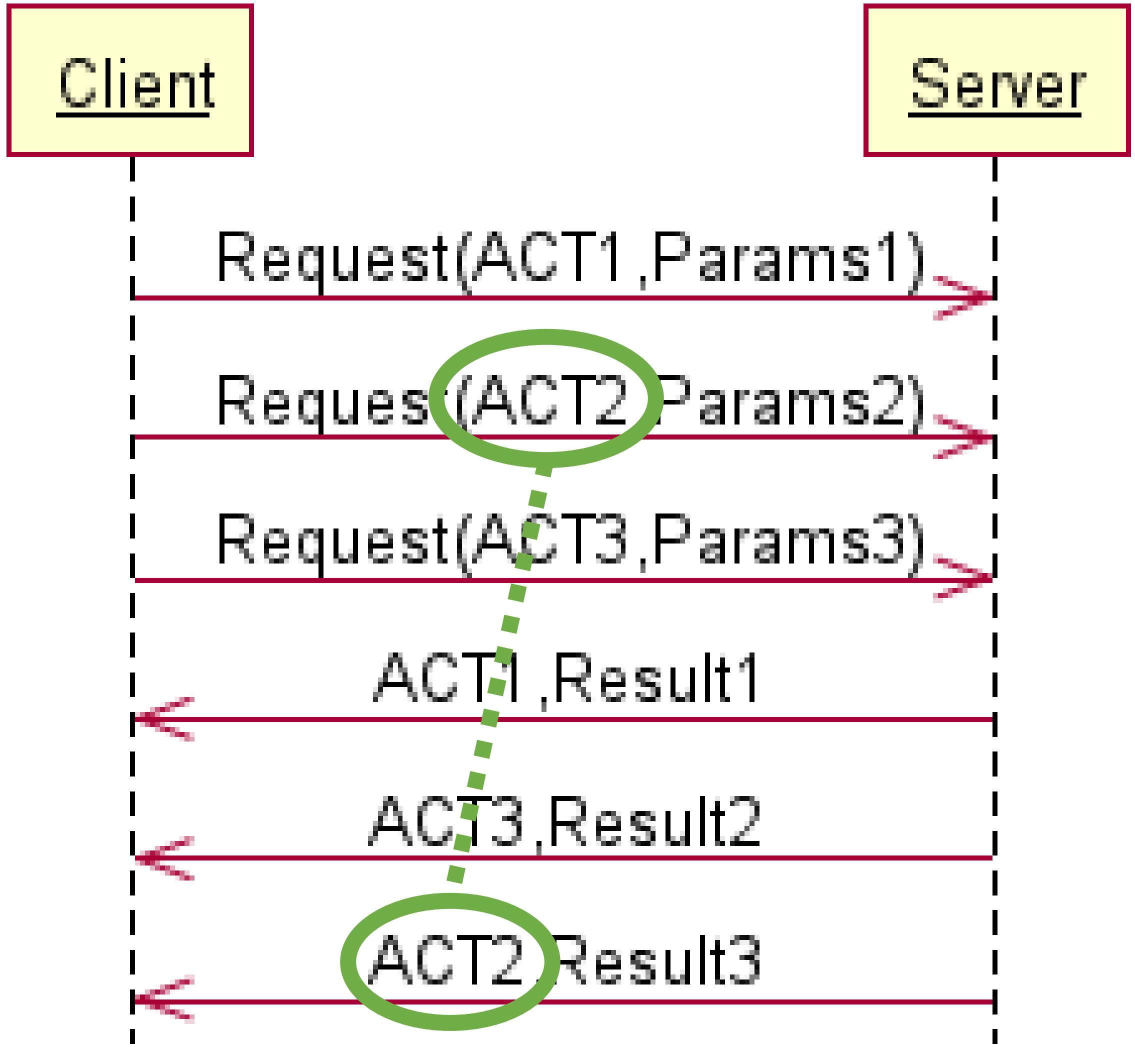
Request/event handling patterns

- Handling synchronous and asynchronous requests
- Input-output handling:
  - blocking IO: read/write waits until completed, calling thread blocks
  - non-blocking, synchronous IO: read/write returns immediately, either with the data read/written or with a signal that the IO operation could not be completed
  - non-blocking, asynchronous IO: read/write returns immediately, operation is started on a background thread, the caller is notified when the operation is ready
- Patterns:
  - Asynchronous completion token
  - Cancellation token
  - Future/Task/Deferred (async-await)

Asynchronous completion token (ACT)

- Problem: client calls multiple asynchronous operations on the server and receives responses for them but not necessarily in order
- The ACT pattern allows efficient demultiplexing responses of asynchronous operations
- ACT pattern:
  - the ACT is usually an identifier
  - client passes the ACT in asynchronous requests
  - server returns the original ACT in asynchronous responses
  - based on the returned ACT the client can identify which response is for which request

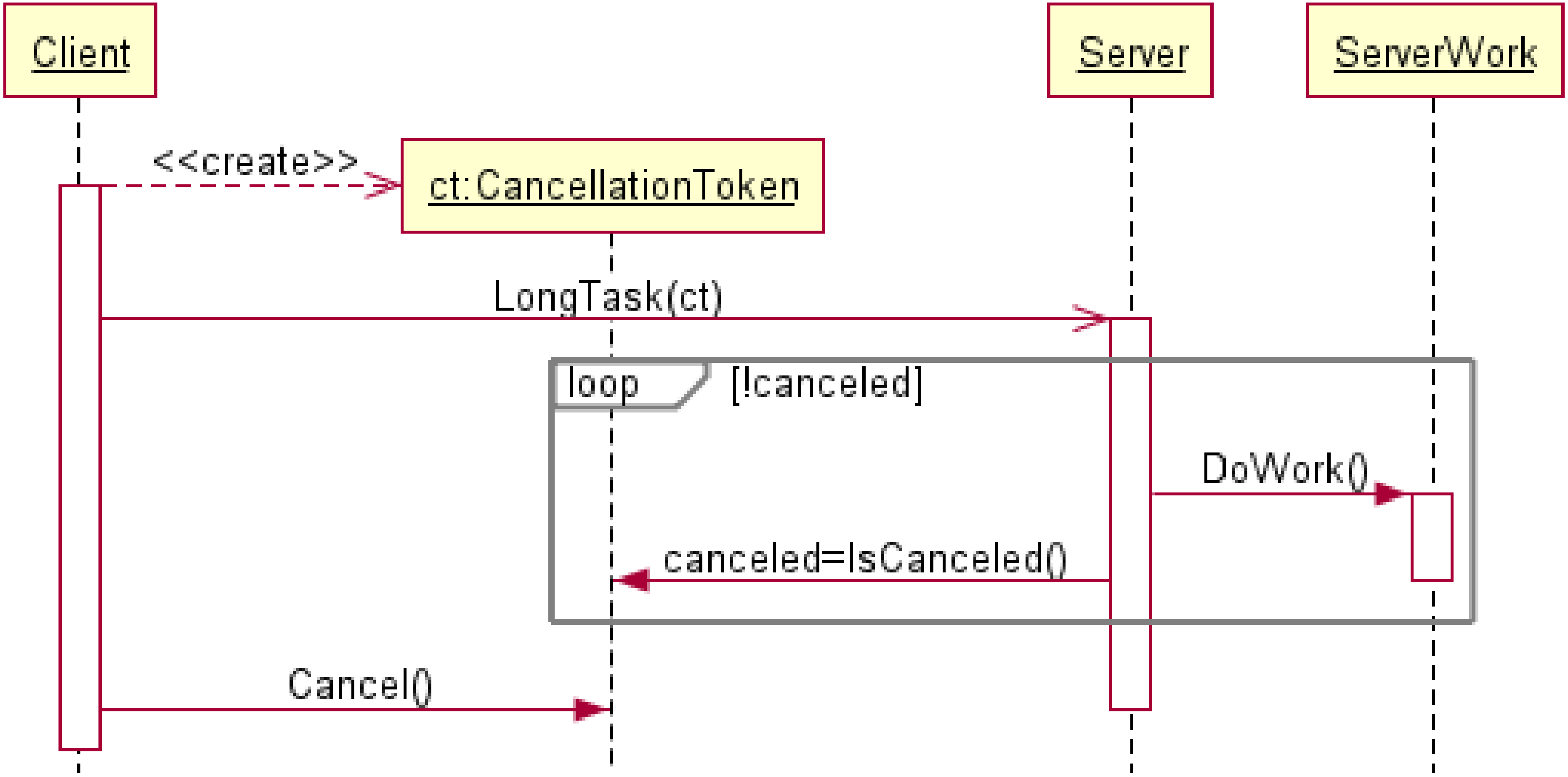
Asynchronous completion token (ACT)



Cancellation token

- Problem:
  - the client calls a long running asynchronous operation on the server frequently
  - the long running operation may not finish before it is called again
  - the result of the old operation becomes irrelevant when the operation is called again
  - the client needs a way to cancel the old operation
  - example:
    - client is the IDE editor, while the programmer is editing source code
    - server is the IDE compiler, which is executed regularly
- Cancellation token pattern:
  - the cancellation token is an object passed to the long running asynchronous operation
  - it can be cancelled by the client
  - the long running asynchronous operation regularly checks the token whether it has been cancelled
  - when the token was cancelled the long running asynchronous operation exits
- .NET:
  - read-only view: System.Threading.CancellationToken class
  - manipulation: System.Threading.CancellationTokenSource class
- Java: no such solution, but can be implemented easily

Cancellation token



Cancellation token example

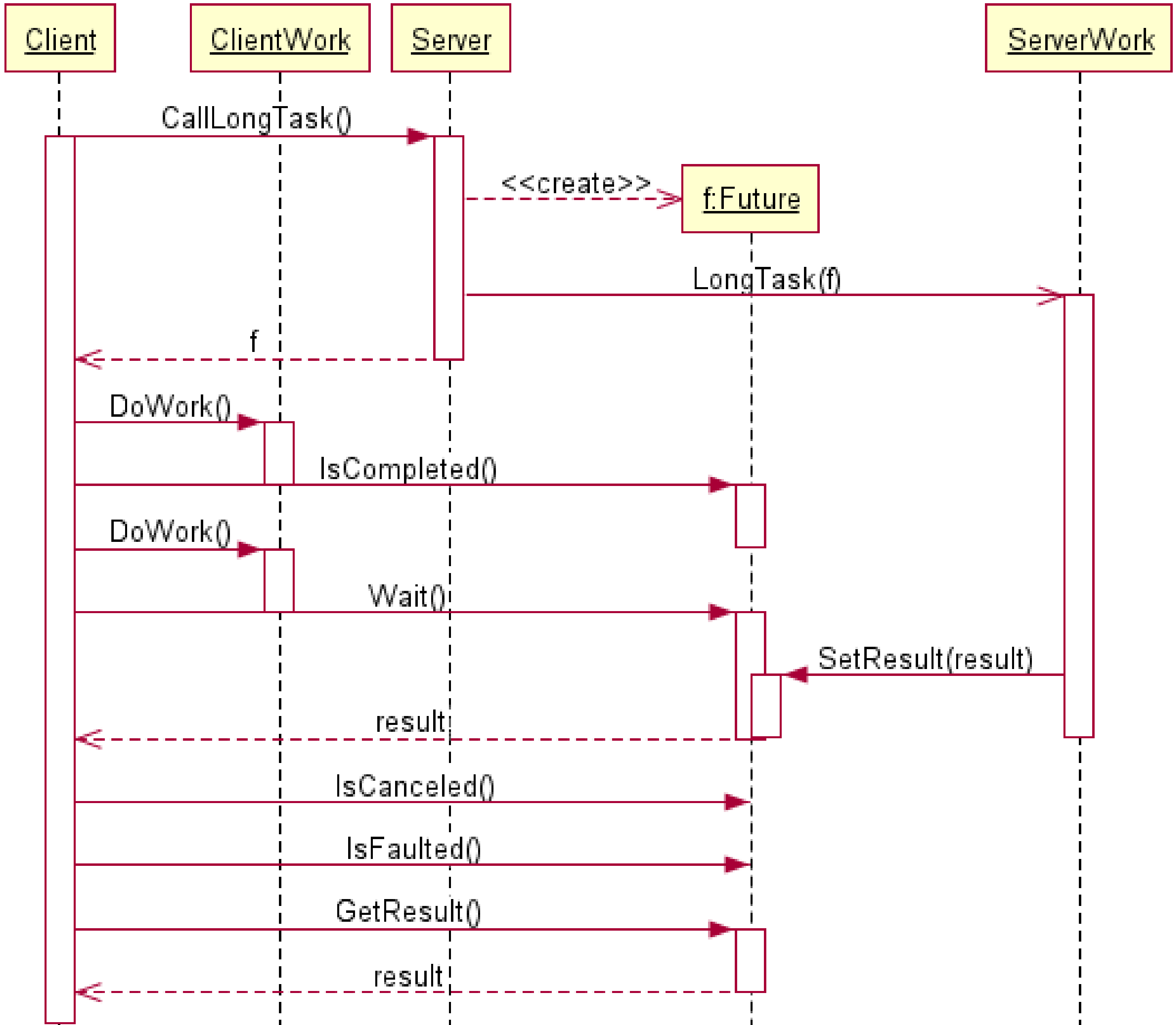
```
public class Client {
    private CancellationTokenSource tokenSource = new CancellationTokenSource();
    private Server server = new Server();

    public void OnEdit() {
        tokenSource.Cancel(); // Cancel previous compilation
        tokenSource = new CancellationTokenSource();
        Task.Run(() => server.Compile(tokenSource.Token));
    }
}
```

```
public class Server {
    public Task<bool> Compile(CancellationToken token) {
        this.DoWork();
        // Return if the operation is cancelled:
        if (token.IsCancellationRequested) return Task.FromResult(false);
        this.DoMoreWork(token);
        return Task.FromResult(true);
    }
    private void DoMoreWork(CancellationToken token) {
        this.DoWork();
        // Return from a deep call by throwing OperationCanceledException
        // if the operation is canceled:
        token.ThrowIfCancellationRequested();
        this.DoWork();
    }
}
```

Future/Task/Deferred

- Problem: client starts an asynchronous operation and needs to access or to synchronize for the completion of the result
- Future/Task/Deferred:
  - returned by the asynchronous operation
  - a read-only view of the execution of the asynchronous operation
  - shows whether the operation is running, completed or cancelled
  - allows waiting for the operation to complete and getting the result
- Manipulating the state of a read-only Future/Task/Deferred is usually in a separated class: Promise
  - resolve/bind the result when the operation is completed
  - cancel the operation
- .NET:
  - read-only view: System.Threading.Tasks.Task<T> class
  - manipulation: System.Threading.Tasks.TaskCompletionSource<T> class
  - C# also has built-in language constructs for tasks: async/await
- Java:
  - almost read-only (allows cancellation): java.util.concurrent.Future<T> interface
  - manipulation: java.util.concurrent.FutureTask<T> implementation of Future<T>
  - unfortunately no clear separation of concerns

Future/Task/Deferred



C# example: Task

```
public void CallLongTaskAndDoSomeWork() {
    CancellationTokenSource ct = new CancellationTokenSource();
    Task<CalculationResult> task = this.CallLongTask(ct.Token);
    this.DoWork();
    if (task.IsCompleted) { } // Check if the task is completed
    this.DoWork();
    task.Wait(); // Wait for the task to finish
    CalculationResult result = task.Result; // Get the result
    if (task.IsCanceled) { } // Check if the task is canceled
    if (task.IsFaulted) { } // Check if the task has thrown an exception
}

public Task<CalculationResult> CallLongTask(CancellationToken token) {
    // Cancellation token is optional, included only for the sake of this example:
    Task<CalculationResult> task =
        new Task<CalculationResult>(this.LongTask, token);
    task.Start(); // Start the task in the background
    return task;
}
public CalculationResult LongTask() {
    CalculationResult result = new CalculationResult();
    this.DoWork();
    return result;
}
```

Java example: Future

```java
private static final ExecutorService threadpool =
    Executors.newFixedThreadPool(3);
public void callLongTaskAndDoSomeWork() {
    Future<CalculationResult> future = this.callLongTask();
    this.doWork();
    if (future.isDone()) { } // Check if the task is completed
    this.doWork();
    future.cancel(true); // Cancel the task
    this.doWork();
    CalculationResult result = future.get(); // Wait for the task to finish
    if (future.isCancelled()) { } // Check if the task is canceled
    // To check exceptions: override FutureTask
    //     or wrap the task to catch and store exceptions
}
public Future<CalculationResult> callLongTask() {
    FutureTask<CalculationResult> task =
        new FutureTask<CalculationResult>(this::longTask);
    threadpool.execute(task); // Start the task in the background
    return task;
}
public CalculationResult longTask() {
    CalculationResult result = new CalculationResult();
    this.doWork();
    return result;
}
```

Summary

Summary

- Distributed OO
- Concurrency problems
- Possible solutions:
  - Synchronization patterns
  - Context patterns
  - Request/event handling patterns