Runtime Execution

Instancing, threading, and throttling

Outline

- **Instancing**
  - PerCall
  - Single
  - PerSession
  - Durable
- **Throttling**
  - ServiceThrottlingBehavior
- **Concurrency**
  - Single
  - Reentrant
  - Multiple

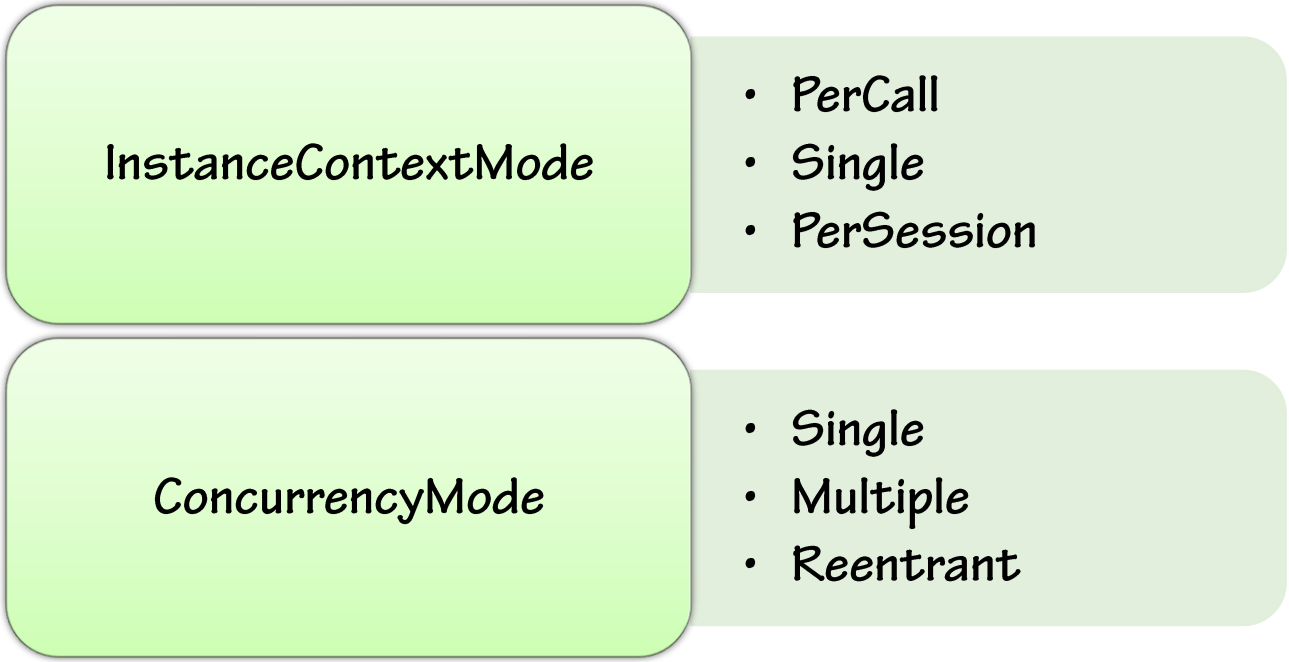
WCF runtime execution

- **Most applications differ widely in their requirements and needs**
  - Scalability, performance, throughput, etc
  - Hence, a single runtime strategy doesn't work for everyone
- **WCF provides the ability to configure key runtime behaviors**
  - Service instancing
  - Throttling
  - Concurrency

these behaviors are mostly a service-side implementation detail

Instancing & threading

- **You control service instancing and threading with [ServiceBehavior]**
  - Service-wide setting, applies to any ServiceHost, all endpoints etc



InstanceContextMode

- **InstanceContextMode controls the service instancing behavior**
  - How the dispatcher manages instance lifecycle (creation and disposal)

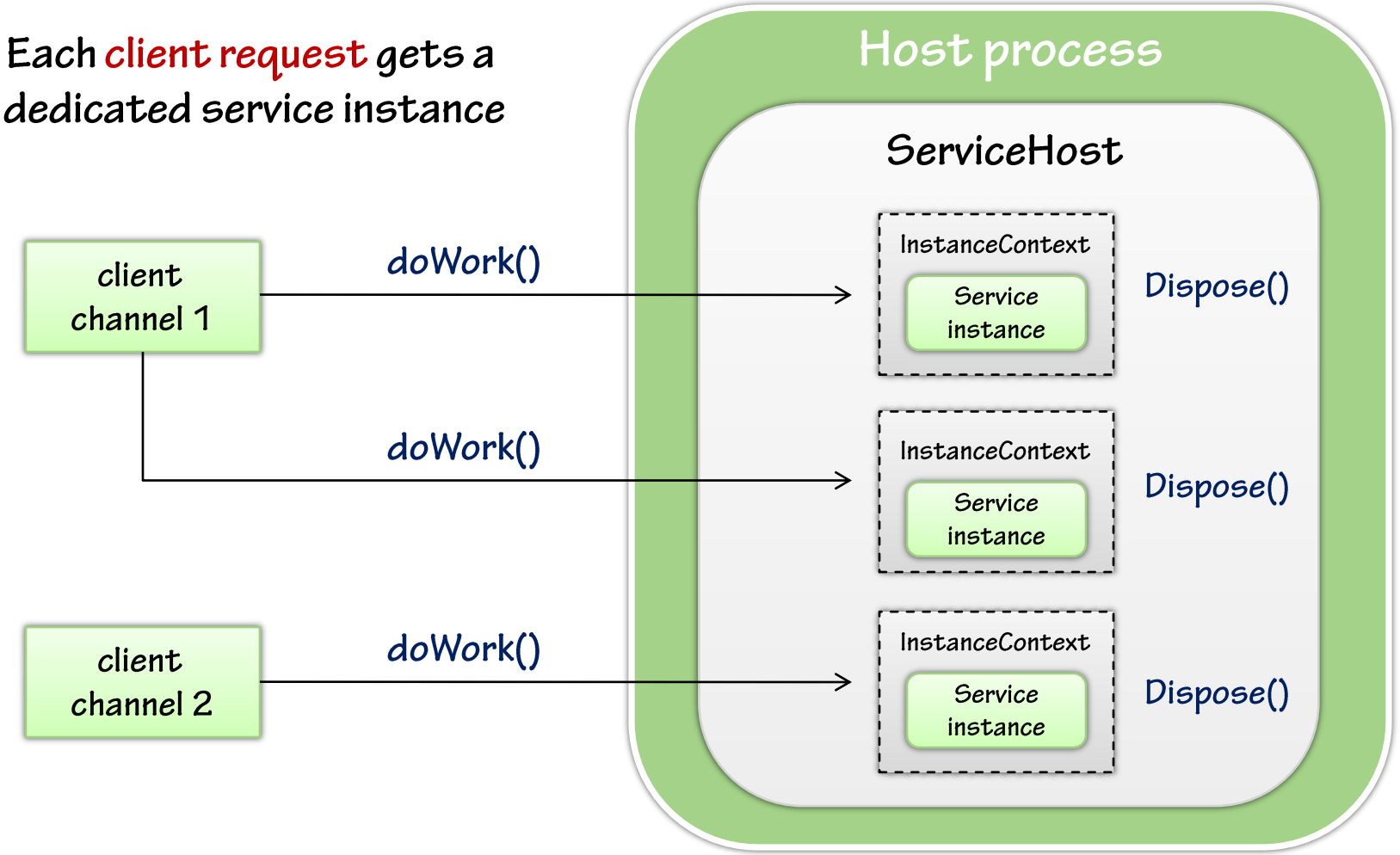
InstanceContextMode	Description
Single	A single service instance is used for all calls
PerCall	A new service instance is created and disposed on each call
PerSession (default)	A new service instance is created for each client channel

InstanceContextMode.PerCall

- **A new service instance is created for each client invocation**
  - Each client request always gets a dedicated service instance
  - No need to worry about concurrent access (always single-threaded)
- **After dispatching is complete, WCF disposes of the instance**
  - WCF calls Dispose if the service implements IDisposable
  - Dispose called on same thread and OperationContext is available
  - After Dispose, WCF releases instance for garbage collection

InstanceContextMode.PerCall

Each client request gets a dedicated service instance



InstanceContextMode.PerCall

Pros

- Simple to use and highly scalable
- Works well in server farms with load balancing
- Only holds resources while actually in use

Cons

- Clients don't know about new instances
- Services must be designed to manage state
- You trade some performance for scalability (cost of creating & disposing instances)

InstanceContextMode.Single

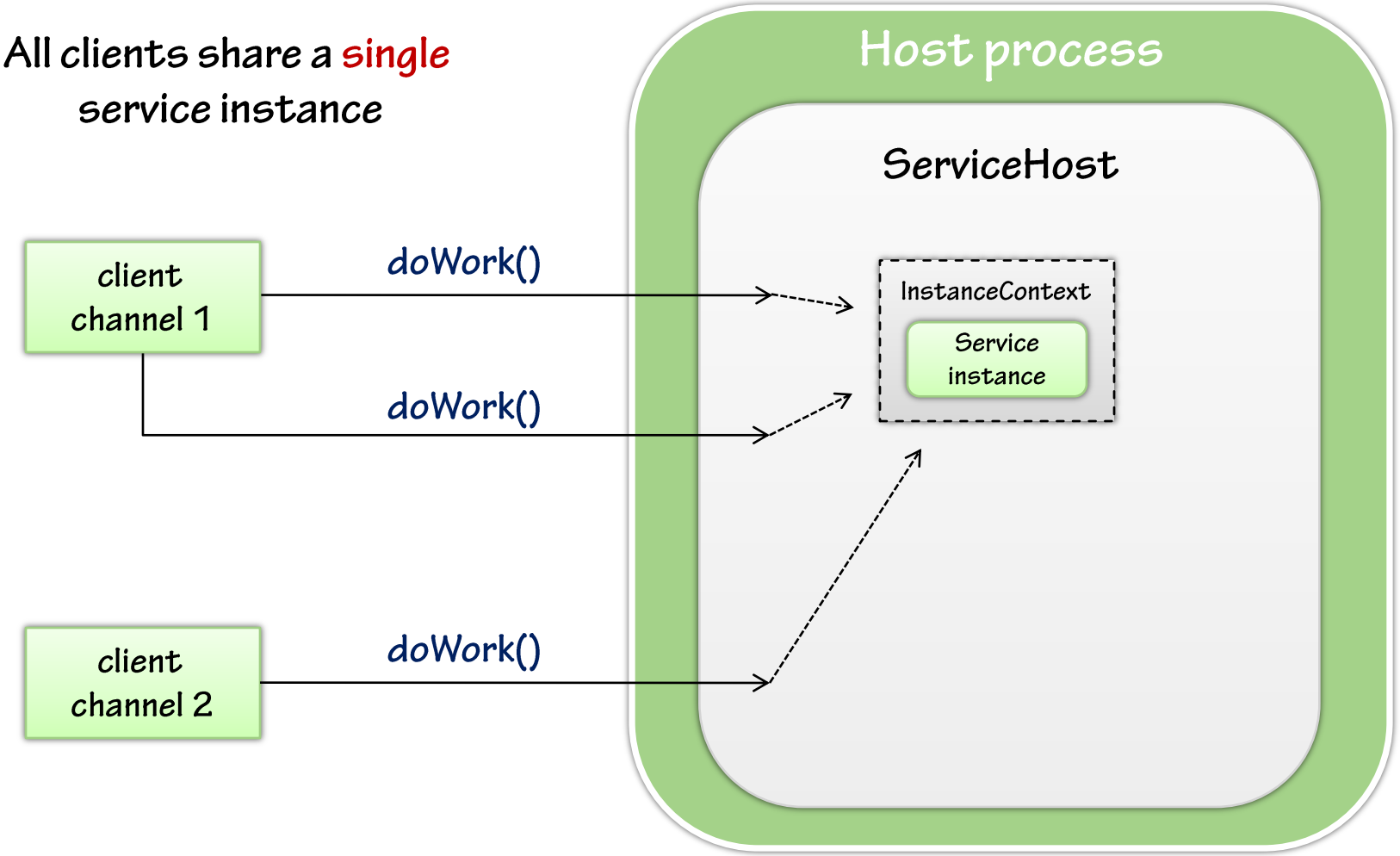
- **All client requests are dispatched to a single service instance**
  - Singleton instance created when ServiceHost is created
  - Or you can provide an initialized instance when constructing ServiceHost
  - All clients share instance, so you must manage concurrency
- **Singleton instance isn't disposed until ServiceHost shuts down**

provide initialized instance to ServiceHost →

```
InvoiceService singleton = new InvoiceService();
... // initialize instance here
ServiceHost host = new ServiceHost(instance);

... // access singleton instance later on
singleton = host.SingletonInstance;
```

InstanceContextMode.Single

All clients share a single service instance



InstanceContextMode.Single

Pros

- Simple model for managing global state
- Good for modeling real "singletons" in the physical world (e.g., a piece of hardware)

Cons

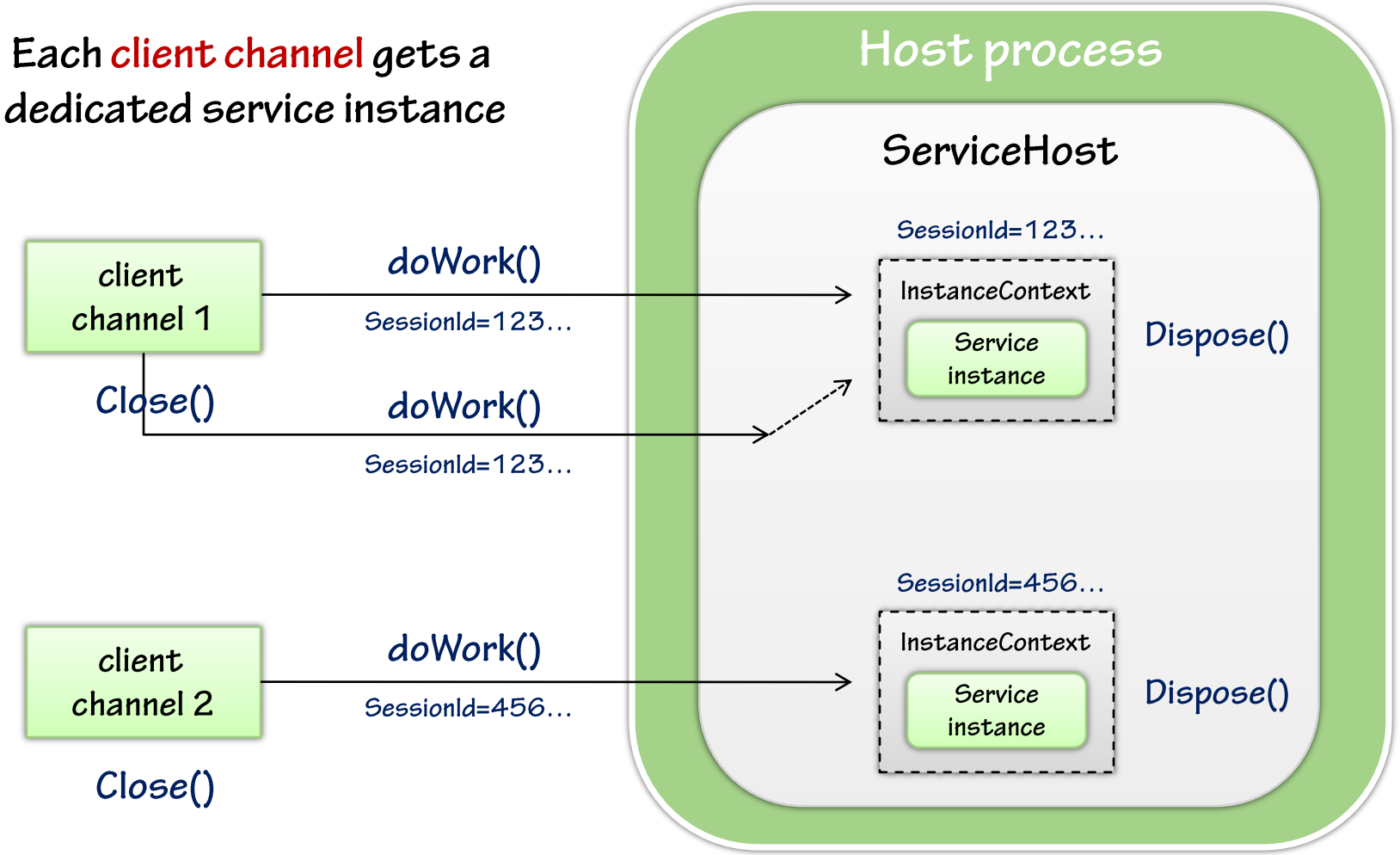
- Very difficult to scale (when updating, only 1 client at a time can access)
- You must manage concurrency
- You must manage session state (if you care)

InstanceContextMode.PerSession

- **Each client channel gets a dedicated service instance**
  - Allows you to manage client state in member variables
- **Session management is provided by the underlying channel stack**
  - You must choose a binding that provides this feature (not all do)
  - If the channel doesn't provide a session, you get PerCall behavior
- **Lifetime of session is typically controlled by the client channel**
  - Once closed, channel notifies service that session has ended
  - WCF calls Dispose if the service implements IDisposable
  - Dispose called on new thread, OperationContext not available
  - Sessions can also time-out after a (configurable) period of inactivity

InstanceContextMode.PerSession

Each client channel gets a dedicated service instance



Bindings that support sessions

- **NetTcpBinding & NetNamedPipeBinding naturally support sessions**
  - WCF associates transport connection with client
- **Sessions can be enabled via SOAP over connection-less protocols**
  - Accomplished using various WS-* specifications
  - WSHttpBinding supports sessions if security is turned on (default)
  - You can also enable RM on the binding to get session support
  - Sessions may or may not be reliable depending on configuration
- **If the channel supports sessions, you'll get a session identifier**
  - Look in OperationContext.Current.SessionId to discover it
  - Useful with both PerSession and Single instancing modes

Demanding a sessionful binding

- **PerSession services are typically designed with that mode in mind**
  - Hence, it probably won't work if the binding doesn't provide sessions
- **You can require sessionful binding using SessionMode**
  - Provides three settings: Allowed (default), NotAllowed, Required
  - Host throws exceptions when mismatches are identified

```
[ServiceContract(SessionMode= SessionMode.Required)]
public interface IInvoiceService { ... }
```

always use SessionMode.Required when designing sessionful contracts

Session termination

- **Sessions typically end when the client closes the channel**
  - But what happens when the client terminates ungracefully?
  - Or what if the network goes down?
- **Sessions have an idle timeout that you can configure**
  - Default value is 10 minutes
  - Clients and services can set this independently (shortest value wins)

set idle timeout to 30 minutes

```
<configuration>
  <bindings>
    <netTcpBinding>
      <binding name="MyConfig">
        <reliableSession enabled="true"
          inactivityTimeout="00:30:00"/>
      </binding>
...
```

Shaping sessions with contracts

- **Sessionful contracts can specify session start/end operations**
  - Use IsInitiating/IsTerminating properties on [OperationContract]
  - Requires SessionMode=SessionMode.Required
  - Useful with both PerSession and singleton services

Property	True	False
IsInitiating	(default) starts new session if it's called first by client	can't be called first by client
IsTerminating	session terminates once the operation returns	(default) session continues after operation returns

Shaping sessions example

allowed to begin session, doesn't end session

not allowed to begin session, may only continue, doesn't end session

terminates session

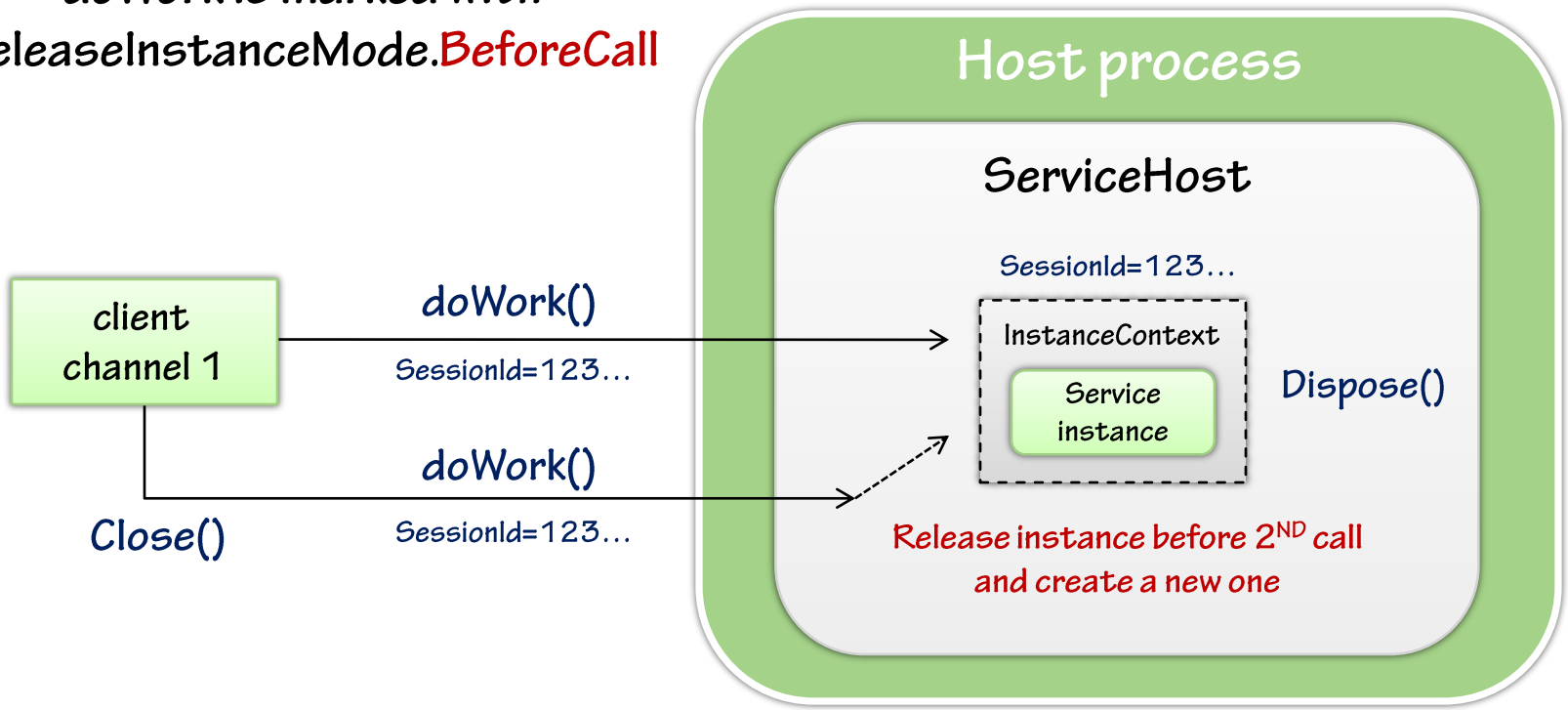
```
[ServiceContract(
   SessionMode=SessionMode.Required)]
public interface IMath
{
    [OperationContract(
      /* IsInitiating=true,
         IsTerminating=false */)]
    void Clear();

    [OperationContract(
         IsInitiating=false
      /* IsTerminating=false */)]
    void Add(int value);

    [OperationContract(
         IsInitiating=false,
         IsTerminating=true)]
    int GetFinalSum();
}
```

Advanced instance management

- **The dispatcher actually correlates each session to an InstanceContext**
  - Hence, InstanceContextMode controls lifetime of the InstanceContext
  - You can separately control instance deactivation within a context
- **Declarative instance deactivation**
  - Use OperationBehavior.ReleaseInstanceMode (None, BeforeCall, AfterCall)
- **Explicit instance deactivation**
  - Use InstanceContext.ReleaseServiceInstance within an operation
- **You can even provide custom instance management with a behavior**

PerSession with ReleaseInstanceMode

doWork is marked with ReleaseInstanceMode.BeforeCall



InstanceContextMode.PerSession

Pros

- Simple model for managing state
- Works great for tightly-coupled systems
- Gives DCOM/.NET Remoting a path forward

Cons

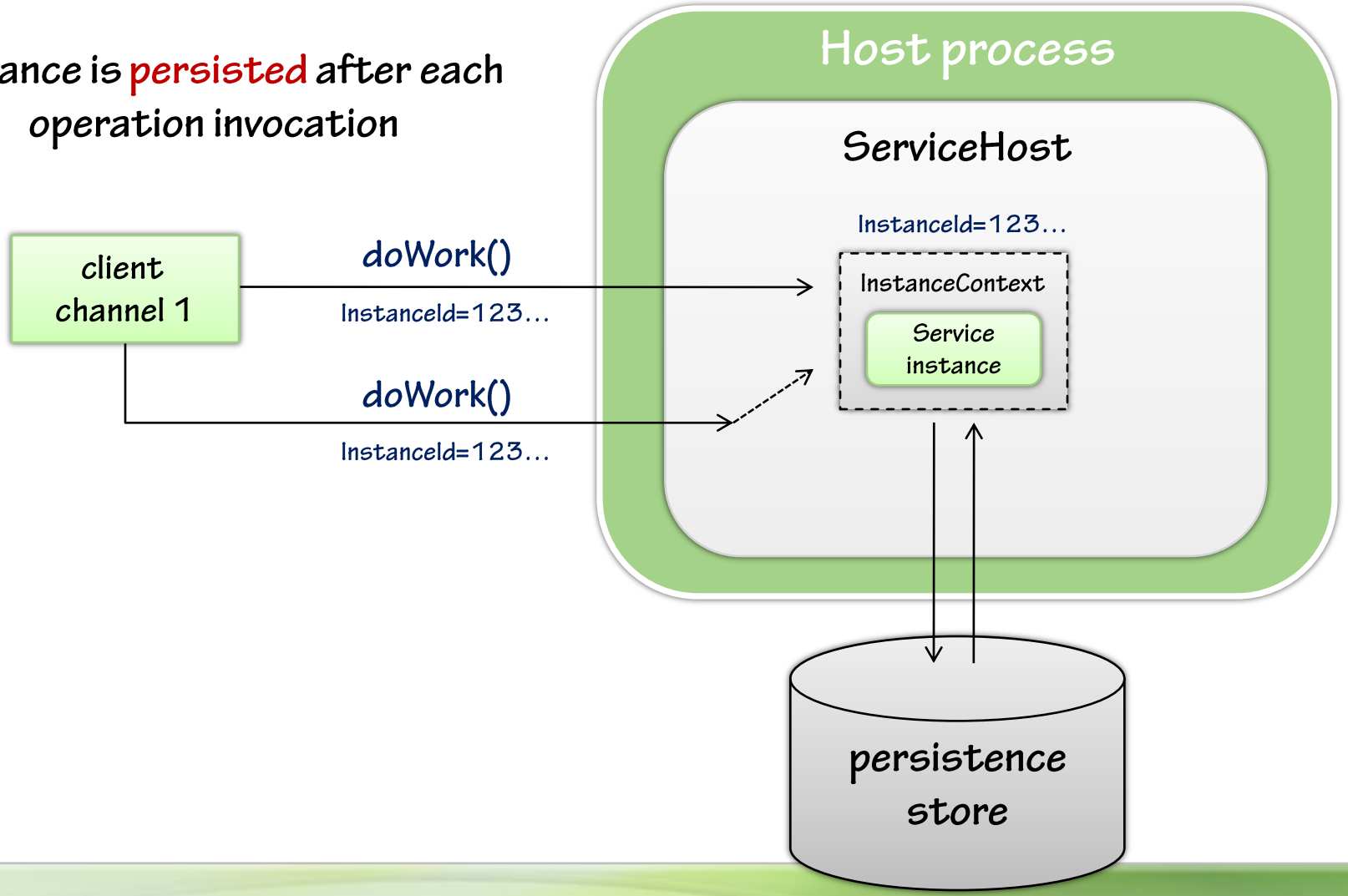
- Limits scalablity
- Ties each client to a particular server machine
- Reduces effectiveness of load balancing

Durable services

- **WCF 3.5 provides a new mechanism for persisting service instances**
  - Like PerSession, only the instance is durable for the session
- **Enabling durable services requires several changes**
  - Annotate the service with [DurableServiceBehavior]
  - Annotate your operations with [DurableServiceOperationBehavior]
  - Configure the service with a PersistenceProviderBehavior
  - Use one of the new "context" bindings that come with 3.5
- **The SDK comes with scripts for the SqlPersistenceProvider**
  - Look in C:\windows\microsoft.net\Framework\v3.5\SQL\EN
- **Clients can resume durable services assuming they save the context**

Durable services

instance is persisted after each operation invocation



Durable services example

```
[Serializable]
[DurableService]
public class DurableCalc
    : ICalculator {
     int sum;
     [DurableOperation]
     public void Add(int value)
         this.sum += value;
     }
     [DurableOperation(
       CompletesInstance = true)]
     public int GetFinalSum() {
         return this.sum;
     }
}
```

```
...
<serviceBehaviors>
  <behavior name="MakeDurable">
    <persistenceProvider
       type="...SqlPersistenceProviderFactory..."
       connectionStringName="SqlPersistenceConnection"
       persistenceOperationTimeout="00:00:10"
       lockTimeout="00:01:00"
       serializeAsText="true"/>
...
```

```
...
<service behaviorConfiguration="MakeDurable"
         name="DurableCalc">
  <endpoint address=""
            binding="basicHttpContextBinding"
            contract="ICalculator" />
...
```

Service throttling

- **WCF allows you to throttle the load on a particular service type**
  - Maximum number of concurrent sessions
  - Maximum number of concurrent calls
  - Maximum number of concurrent instances
  - Configured using the ServiceThrottlingBehavior
- **When exceeded, WCF places pending callers in a queue**
  - Pending callers dispatched from queue in order
  - Clients could timeout depending on setting
- **Some bindings also allow you to constrain max connections**
  - NetTcpBinding & NetNamedPipeBinding

Throttling examples

set custom throttling settings and apply to service (not shown)

throttle a specific binding configuration

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="Throttling">
          <serviceThrottling maxConcurrentCalls="20"
                             maxConcurrentSessions="50"
                             maxConcurrentInstances="100"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <bindings>
      <netTcpBinding>
        <binding name="CustomTcp" maxConnections="50"/>
      </netTcpBinding>
    </bindings>
...
```

ConcurrencyMode

- **All incoming WCF requests are executed on CLR worker threads**
  - You can control how they're allowed to interact with your instances
- **ConcurrencyMode controls the service threading behavior**
  - It basically defines how WCF takes locks before calling your service
- **When using PerCall instancing, you don't have a choice**
  - PerCall instances are naturally single-threaded

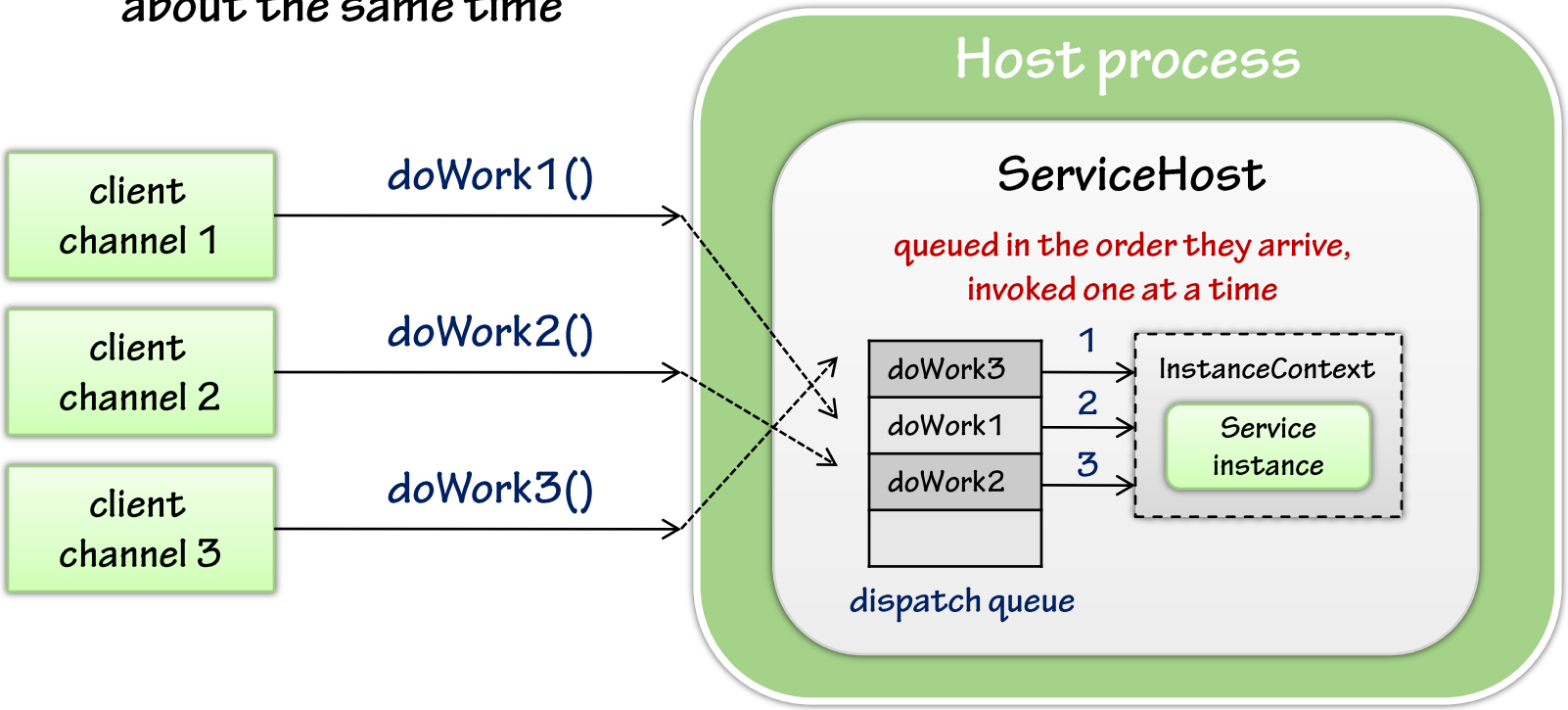
ConcurrencyMode	Description
Single (default)	Dispatcher grabs a lock before entering your code
Multiple	WCF threads don't grab any locks before entering your code
Reentrant	Similar to Single but also allows incoming calls while your service is blocked on an outgoing call (avoids deadlock)

ConcurrencyMode.Single

- **Only one thread at a time will be allowed to call the service instance**
  - Each service instance is associated with a synchronization lock
  - WCF attempts to acquire lock before each invocation
  - If lock is already owned, caller is placed in queue and must wait
- **Frees you from dealing with concurrency within your service code**

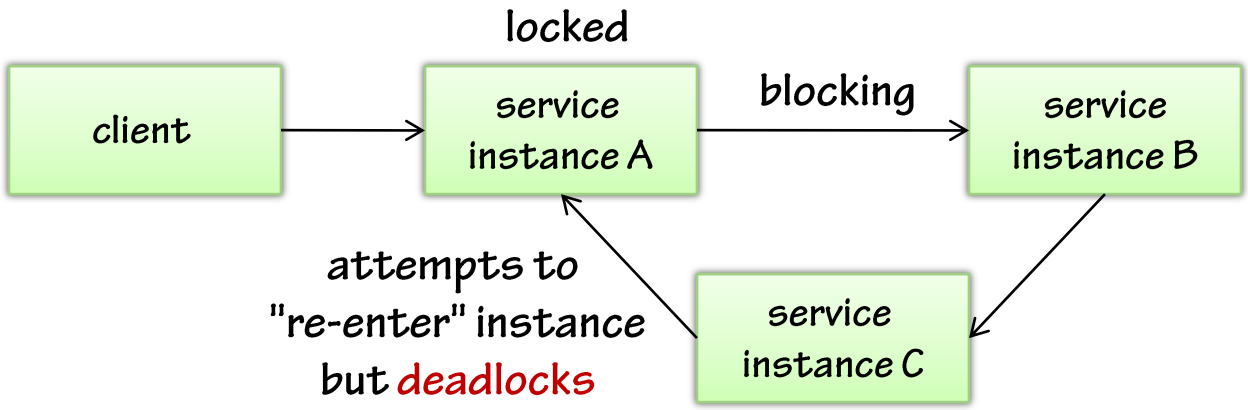
ConcurrencyMode.Single

several simultaneous calls arrive about the same time



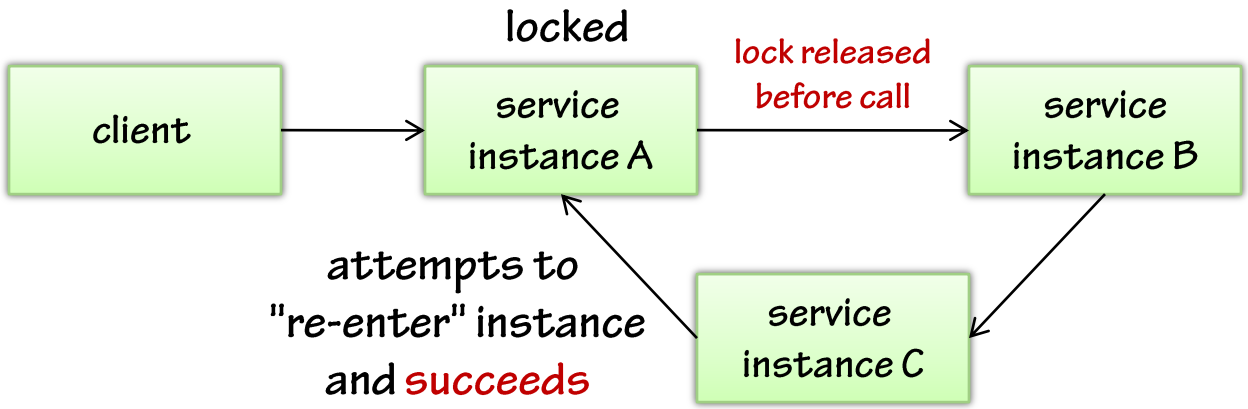
Deadlocks and reentrancy

- **Deadlocks are possible with ConcurrencyMode.Single**
  - Occurs when a downstream instance attempts to re-enter locked instance
  - Timeouts are the only way to break the deadlock



ConcurrencyMode.Reentrant

- **ConcurrencyMode.Reentrant is designed to avoid deadlocks**
  - Like Single, only one thread will touch your service instance at a time
  - Releases lock just before making an outgoing call
  - Future callbacks contend for access like any other request
- **You must design your services carefully to handle reentrancy**
  - Instance must ensure consistent state before/after outbound call

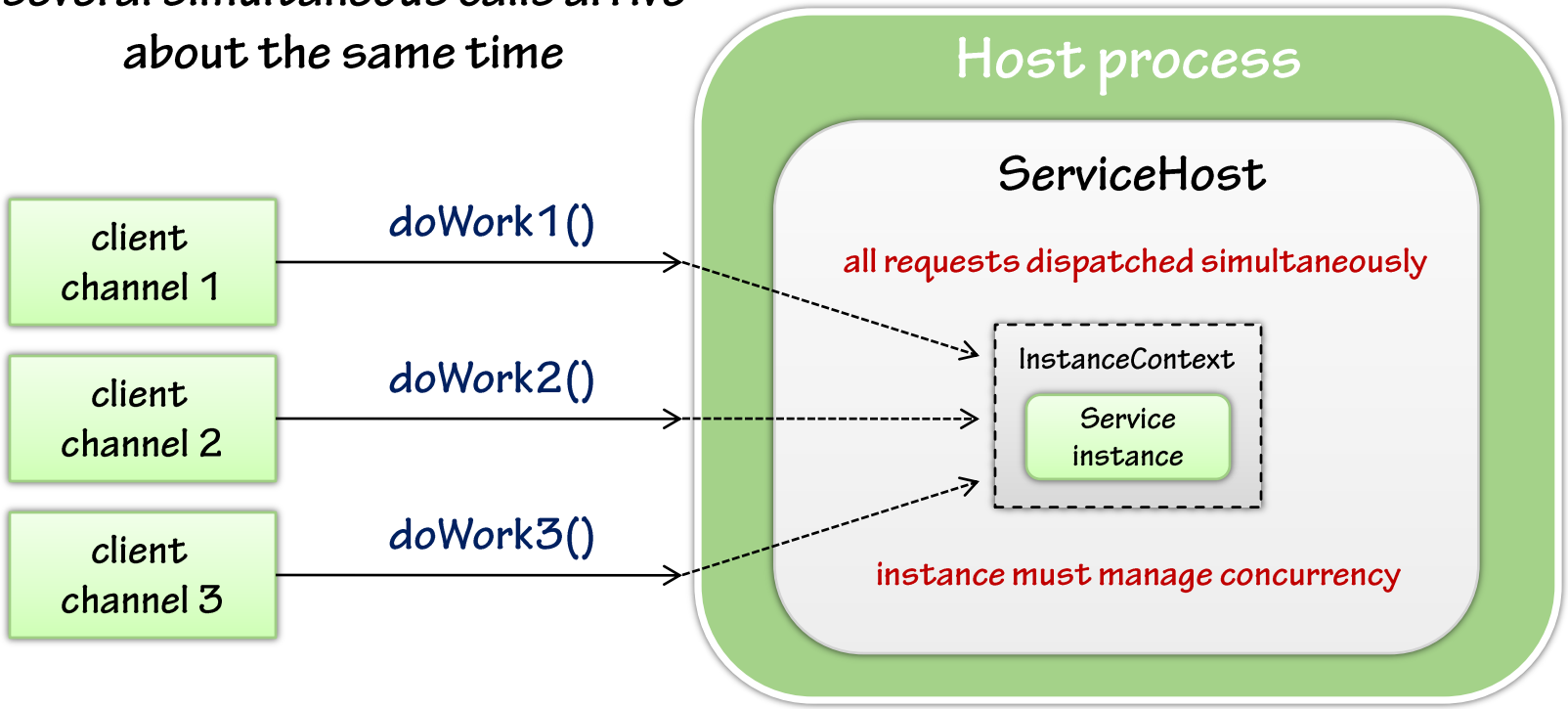


ConcurrencyMode.Multiple

- **WCF is allowed to access the service instance with multiple threads**
  - Service is not associated with a synchronization lock
  - Incoming calls are immediately dispatched (up to throttling limits)
- **You must write synchronization code within service**
  - Use a lock() block to protect specific members or entire service instance
  - Use [MethodImpl] to auto-inject a lock() block around a method body

ConcurrencyMode.Multiple

several simultaneous calls arrive about the same time



Synchronization contexts

- **.NET 2.0 introduced the concept of synchronization contexts**
  - A simple API for marshalling calls across threads (SynchronizationContext)
- **Every thread can have a synchronization context associated with it**
  - Stored in TLS, accessed via SychronizationContext.Current
- **WCF can maintain affinity with the current synchronization context**
  - ServiceBehaviorAttribute.UseSynchronizationContext=true (default)
  - When the host opens, it associates itself with the current context
  - All service instances will execute on that synchronization context

Classic use-case is hosting a WCF service on a UI thread
(so the instance can update the UI controls)

Summary

- **WCF provides some key behaviors for controlling runtime execution**
  - ServiceBehaviorAttribute allows you to control instancing & threading
  - ServiceThrottlingBehavior allows you to limit the service load
- **InstanceContextMode provide three instancing modes**
  - Single, PerCall, and PerSession
- **ConcurrencyMode also provides three threading modes**
  - Single, Reentrant, and Multiple
- **These behaviors allow WCF to adapt to various application scenarios**

References

- **WCF Instance Management**
  - http://msdn.microsoft.com/winfx/technologies/communication/default.aspx?pull=/msdnmag/issues/06/06/wcfessentials/default.aspx
- **MSDN: Sessions, Instancing, and Concurrency**
  - http://msdn2.microsoft.com/en-us/library/ms731193.aspx
- **Pluralsight's WCF Wiki**
  - http://pluralsight.com/wiki/default.aspx/Aaron/WindowsCommunicationFoundationWiki.html