How Delta Propagation Works in Native Client:

In order to propagate deltas, application classes that represent value types must derive from `gemfire::Delta` abstract class publicly, in addition to deriving from `gemfire::Cacheable`. Please see declaration for `class SimpleDeltaExample` in the code snippet. Also, a corresponding java class for each C++/C# class must be registered at the server and peer members of the distributed system. The java class must implement the interfaces `com.gemstone.gemfire.DataSerializable` and `com.gemstone.gemfire.Delta`, and the class must satisfy serialization requirements as a `DataSerializable` type (refer to Section 17.2 '*Data Serialization*' in *GemStone GemFire Enterprise Java Developer's Guide*).

N.B.: When delta propagation is used for a region, only one value type may be used for the region i.e. the value objects for keys in the region must be objects of the same class.

Put operation:
When a put operation is invoked on a key with an updated value, the client library determines if the change in the value object can be represented by a delta by invoking `Delta::hasDelta( )`. If `hasDelta( )` returns true, the delta for value object is obtained by calling `Delta::toDelta()`. The serialized representation of the object's delta is then sent to the server, which applies the change by calling `Delta.fromDelta( )` on the object stored in the region entry for the key.

Get operation:
Get operation works in the same manner when the value types derive from `gemfire::Delta`.

Notifications:
In case a client has subscribed for updates for a key, and delta is applied on the value for the key in a server, the notification received by the client will contain the delta for the key instead of the complete object. Delta is applied to the current value for the key in the client's cache, by invoking `Delta::fromDelta( )` on the value object.

Code snippet of an application class that uses delta propagation:
For a complete example, please refer to the C++/C# QuickStart examples for delta propagation.

C++ code snippet:

```
class SimpleDeltaExample: public Cacheable, public Delta {
  int counter;
  char bytes[1024];
  bool m_hasDelta;

  public:
  // Cacheable methods for serialization
```

```
  virtual int32_t classId( ) const = 0;
  void toData(DataOutput& output) const;
  Serializable* fromData(DataInput& input);

  // Delta methods
  virtual bool hasDelta() { return m_hasDelta; }
  virtual void toDelta(DataOutput& out) const
  {
    out.writeInt(counter);
  }
  virtual void fromDelta(DataInput& in) { in.readInt(counter); }

  //Helper methods
  void setDelta() { m_hasDelta = true; }
}
```

C# code snippet:

```
class SimpleDeltaExample: IGFSerializable, IGFDelta {

  private int counter;
  private char bytes[1024];

  // IGFSerializable methods for serialization
  public void ToData(DataOutput DataOut);
  public IGFSerializable FromData(DataInput DataIn);
  public UInt32 ClassId
  {
    get();
  }

  // Delta methods
  public bool HasDelta();
  public void ToDelta(DataOutput DataOut);
  public void FromDelta(DataInput DataIn);
}
```

Java code snippet:

```
class SimpleDeltaExample implements DataSerializable, Delta {
  // DataSerializable methods
  public void fromData( DataInput in ) throws IOException
  {
  }

  public void toData( DataOutput out ) throws IOException
  {
  }

  // Delta methods
  public boolean hasDelta( )
  {
    // return boolean to indicate whether the object has delta
  }

  public void fromDelta( DataInput in ) throws IOException
```

```
  {
  }

  public void toDelta( DataOutput out ) throws IOException
  {
  }
}
```

Delta Propagation API

This section describes the interfaces, methods and classes for the Delta Propagation API.

This interface defines a contract between the application and Native Client.
The application must define its domain object types by publicly deriving from the classes `Cacheable and Delta` to use the delta propagation feature. (C# classes must implement the `IGFSerializable` and `IGFDelta` interfaces.)
Native Client then determines whether an application object contains a delta, to extract the delta from an application object into `DataOutput`, and to generate a new application object by applying a delta present in the `DataInput` object to an existing application object. The difference in the object state is contained in the `DataOutput` and `DataInput` instances.
Native Client does not provide for versioning, so the application type object must track the changes that are made to the object.

The `class Delta` provides the following public methods:

a. `boolean hasDelta( )`—This method returns a boolean value indicating if the instance contains delta. If hasDelta returns false then full object will be sent. Otherwise delta will be sent.

b. `void toDelta(DataOutput& out)`–This method is invoked at the delta sender-side, on a new application object after Native Client determines the presence of a delta by calling `hasDelta` on the object. The delta is written to the `DataOutput` object provided by Native Client.

c. `void fromDelta(DataInput& in)`–This method is invoked at the delta receiver-side, on an existing application object if the application is aware that the `DataInput` represents a delta.

d. `DeltaPtr clone( )`– This method is invoked when delta is applied via notification for a subscription, on the existing application object. Native Client will call this method to obtain clone of the existing object in order to apply delta and replace it with the existing object in the cache. This facility is there to avoid in place replace of existing data object. Cloning of objects is optional and can be configured with the region attribute ‘cloning-enabled’.
To clone an object for a C# class, the class must implement the `System.ICloneable` interface.

InvalidDeltaException
The `InvalidDeltaException` indicates that an error has occurred while processing a delta at the receiver-side. Native Client catches this exception and re-sends the corresponding full value to the receiver. When the exception is caught while applying delta via notification for a subscription, the full object is obtained by the client from the server.

Delta Propagation Properties

cloning-enabled: a boolean region attribute.

When the region attribute is set to true, deltas for the region are applied to cloned copies of the values and then saved to the cache. When false, the values are modified in place. The default value is true.
Your decision about whether to use cloning will be based on your data consistency requirements, your listener requirements, and your performance needs.
Default implementation of clone method will return same object wrapped in smart pointer.
Without cloning:

a. Data is more likely to end up in an inconsistent state from concurrent applications of delta updates.
b. Listeners that receive delta-related events can't see the old entry value. Native Client modifies the entry in place and so loses its reference to the old value.

Errors in Delta Propagation:

For a put operation in client, if a server receives an `InvalidDeltaException` while applying delta, or is unable to apply delta due to missing entry, the server sends `InvalidDeltaException` in the response message to client. The client then re-sends the full object to the server.

When delta cannot be applied via notification due to missing entry (eg caused by local destroy or expiration), the client requests the full object.

Miscellaneous:

When a client region is cacheless, or conflation is enabled for subscriptions, the server sends only full objects in notifications i.e. deltas are not sent in notifications that are affected by these configurations.