24 February 2010

Peter Kao
Insomniac Games

Introduction to Sync Host

Outline

- Motivation
  - Challenges
  - Solution
- Architecture
  - Objects
  - Authority
- Synchronizing Changes
  - Updates
  - Events and Responses
  - Client API

Outline

- Motivation
  - Challenges
  - Solution
- Architecture
  - Objects
  - Authority
- Synchronizing Changes
  - Updates
  - Events and Responses
  - Client API

Challenges

- Singular authority
  - Authority refers to client that is in charge of state changes on an object
    - If multiple clients try to change state on the same object, they will eventually be in inconsistent states
    - Two players try to take a dropped weapon, at the same time and both end up getting it
  - Multiple clients can request changes to state, but only one client can actually make those changes
    - Multiple players can damage a car, but only one player will change its health

Challenges (continued)

- Choosing an Authority
  - How to get clients to agree on who is to be authority of an object
  - Detect that we need to choose a new authority because
    - Authority not yet chosen
    - Current authority is unresponsive (lag, unclean disconnect)
    - Current authority has disconnected
  - Having clients negotiate amongst themselves is time-consuming and often unreliable

Challenges (continued)

- Late-Join
  - Client is joining a game in-progress and needs to have their state brought up-to-date
  - Who sends updates incoming client; basically, who has the authoritative game state
  - Choosing one client has problems
    - Bandwidth strain on chosen client
    - What if the chosen client lags out?

Outline

- Motivation
  - Challenges
  - **Solution**
- Architecture
  - Objects
  - Authority
- Synchronizing Changes
  - Updates
  - Events and Responses
  - Client API

Solution

- Server to Make Decisions
  - Client always has connection to server – if client looses server connection, then the client gets kicked out of the gam
  - Chooses the authority of objects
  - Has the authoritative (most up-to-date) game state
    - Late-joining clients can ask server for game state update

Solution (continued)

- Differences from a Traditional Server
  - Does not run game-specific logic
  - Maintains a list of clients to pick authority from
  - Database of objects, but does not know what those objects are
    - Each object is just a collection of data
    - Changes to an object's data are routed through the server first so it has authoritative game state
    - Data changes are then forwarded to clients – up to the client to make sense of data changes
  - Data changes are done through messages

Outline

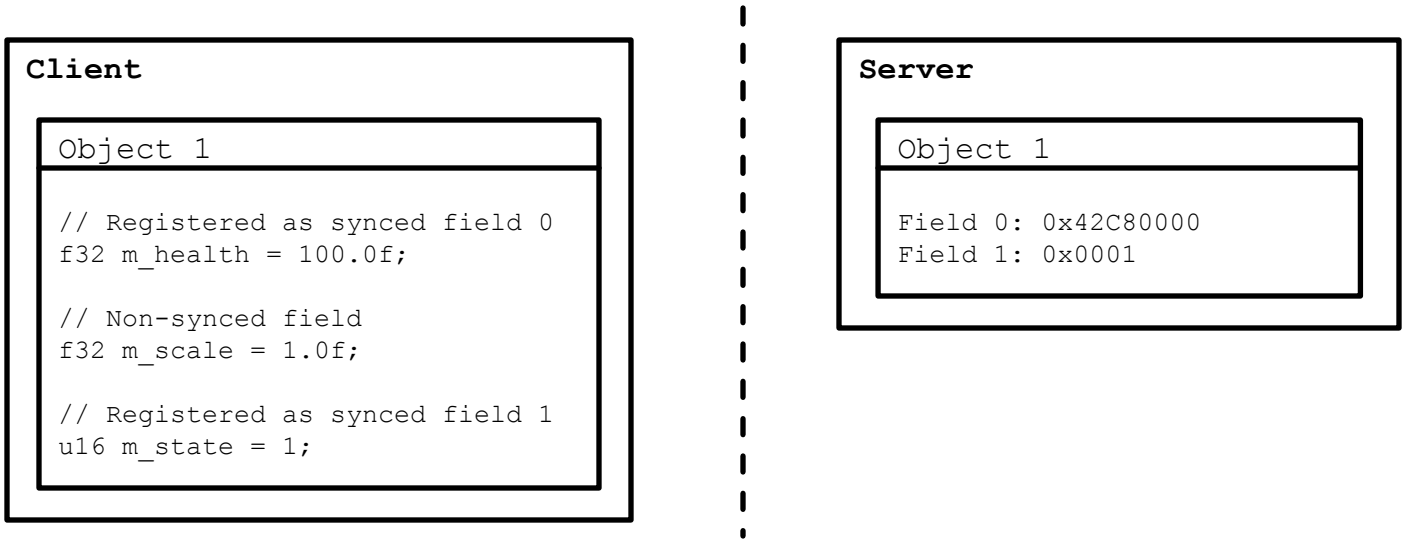
- Motivation
  - Challenges
  - Solution
- Architecture
  - **Objects**
  - Authority
- Synchronizing Changes
  - Updates
  - Events and Responses
  - Client API

Objects

- Client vs. Server Representation
  - Client has familiar view of object (e.g. an update class)
  - Subset of object data is synced – these are called synced fields
    - Each field of an object has its own id
    - Not all fields need to be synced – only things that when changed, other clients need to know about (position, health)
  - Server only knows about synced fields

Objects (continued)

- Client vs. Server Representation

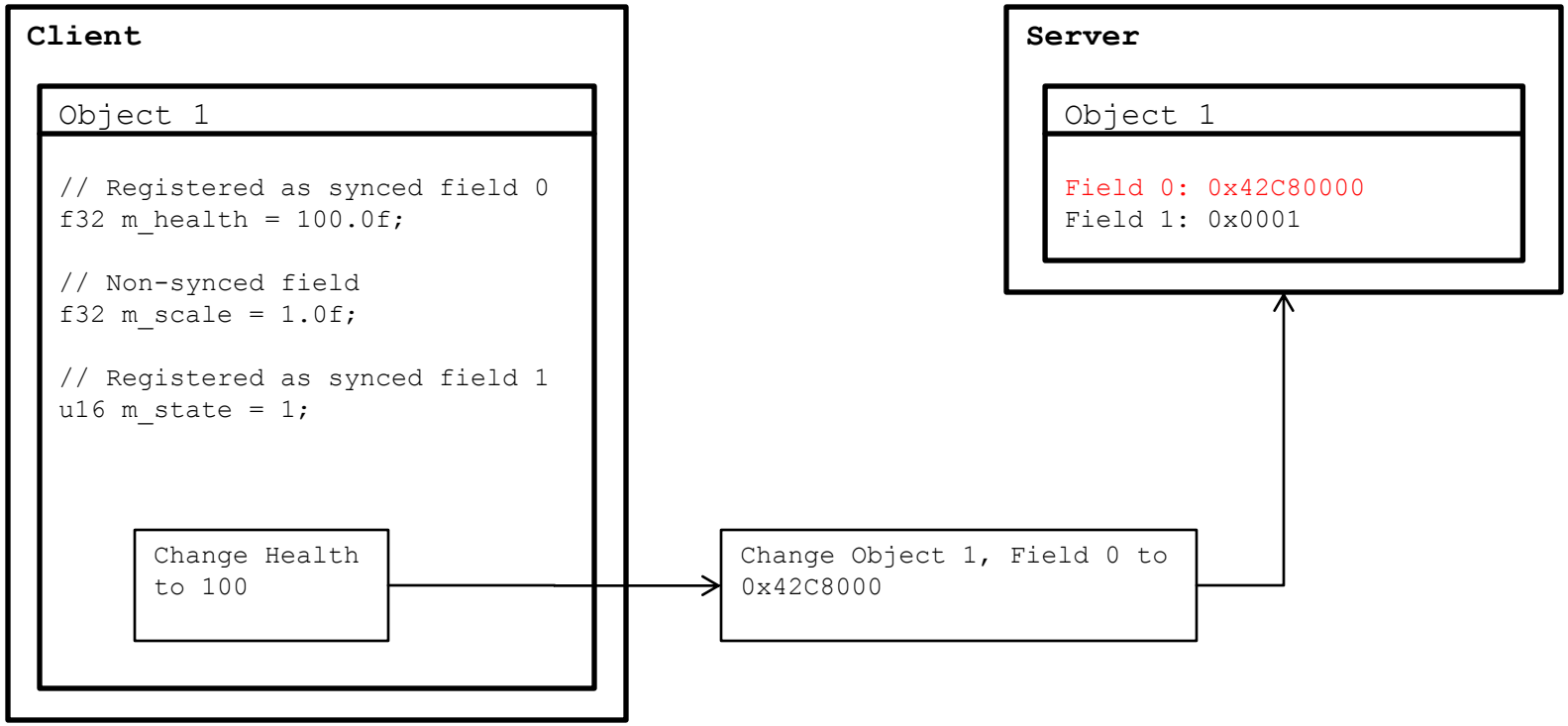


Objects (continued)

- Each object is referenced by a unique object id that is known to the server and each client
  - For example, the same pickup pad on a level will have the same object id on each client
- Assigning object ids
  - For static objects, they can be assigned ahead of time – all pick up pads are loaded in the same order on all clients.
    - Pickup pad 1 gets id 1, pad 2 gets id 2, etc.
  - For dynamic objects like spawned mobys, server has object creation functionality
    - Can ask server to assign an id for an object within a given range

Objects (continued)

- Changing object data



Outline

- Motivation
  - Challenges
  - Solution
- Architecture
  - Objects
  - **Authority**
- Synchronizing Changes
  - Updates
  - Events and Responses
  - Client API

Authority

- Refers to the client that the server has chosen to make state changes about an object
- Two types of authority, default and permanent

Authority (continued)

- Default authority
  - Authority can migrate from one client to another
  - When migration occurs
    - Object does not have an authority yet
    - Current authority is unresponsive or has disconnected
  - A client knows whether or not it's the authority of an object
  - Cannot reliably know who the authority is if it is another client

Authority (continued)

- Permanent authority
  - Once authority client is chosen, it does not change
  - If authority client disconnects, object is deleted
  - Commonly used for player-owned objects (deployable turrets, stat objects)

Authority (continued)

- Authority groups
  - Each object is part of a group and has a group id
  - A group can have multiple objects in it
  - Server assigns the authority of a group to a client, that client becomes authority of all objects in that group
  - Object whose authority is independent of other objects will be in its own group
  - Authority groups are useful when two or more objects should be updated by the same client
    - All bots are usually in same authority group because of the job system and nav reservations

Outline

- Motivation
  - Challenges
  - Solution
- Architecture
  - Objects
  - Authority
- Synchronizing Changes
  - **Updates**
  - Events and Responses
  - Client API

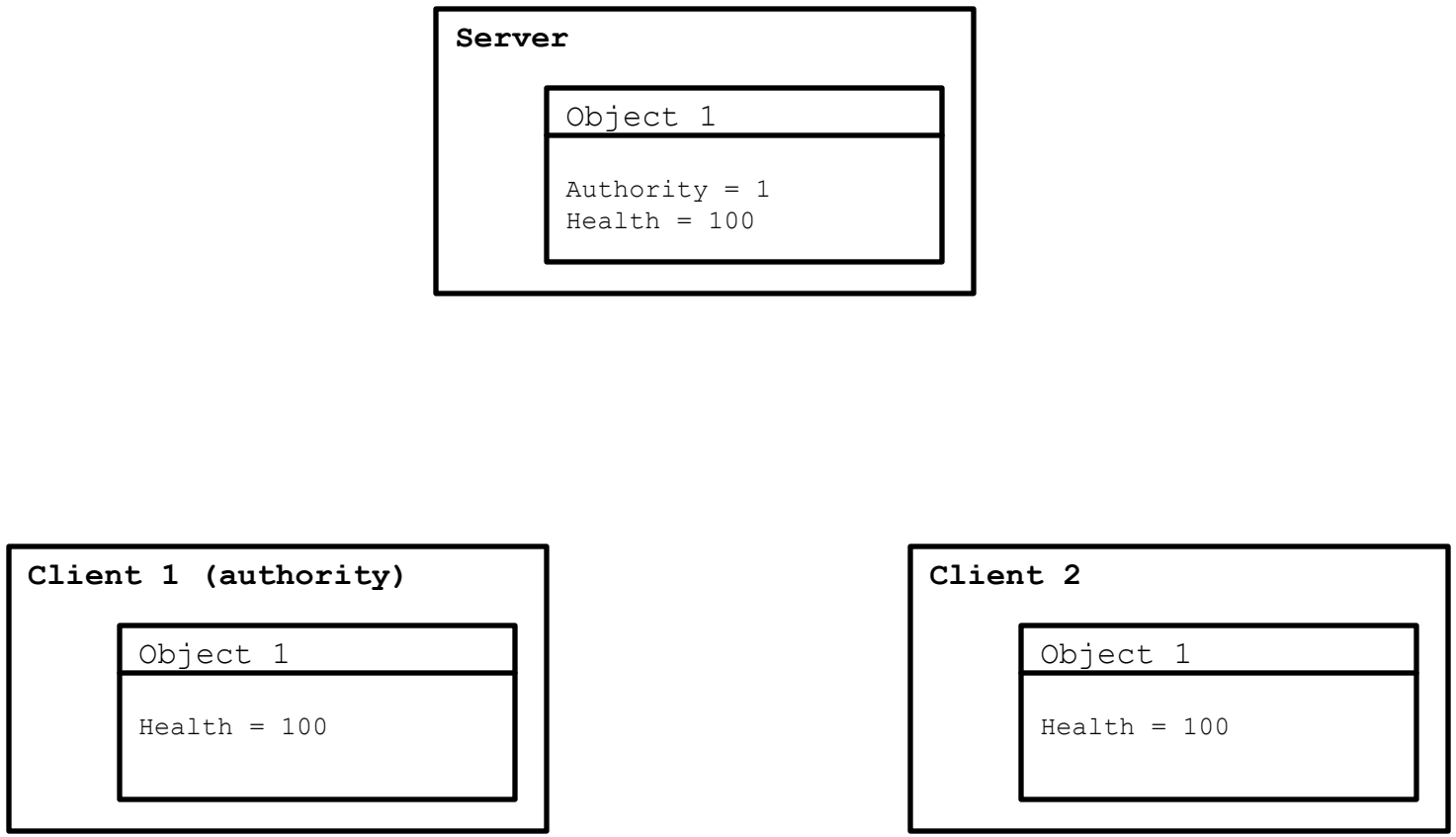
Updates

- Usually used for things that need periodic updates
  - Position changes for a moby, state changes for a bot
  - Server will take update and forward it to clients
- Sent only by the authority, client-side logic usually comprises
  - Check if we're the authority
  - If so, run some logic to determine changes that need to be made (e.g. move position 1 meter)
  - Send changes to server
- If a non-authority sends an update, the server will discard it

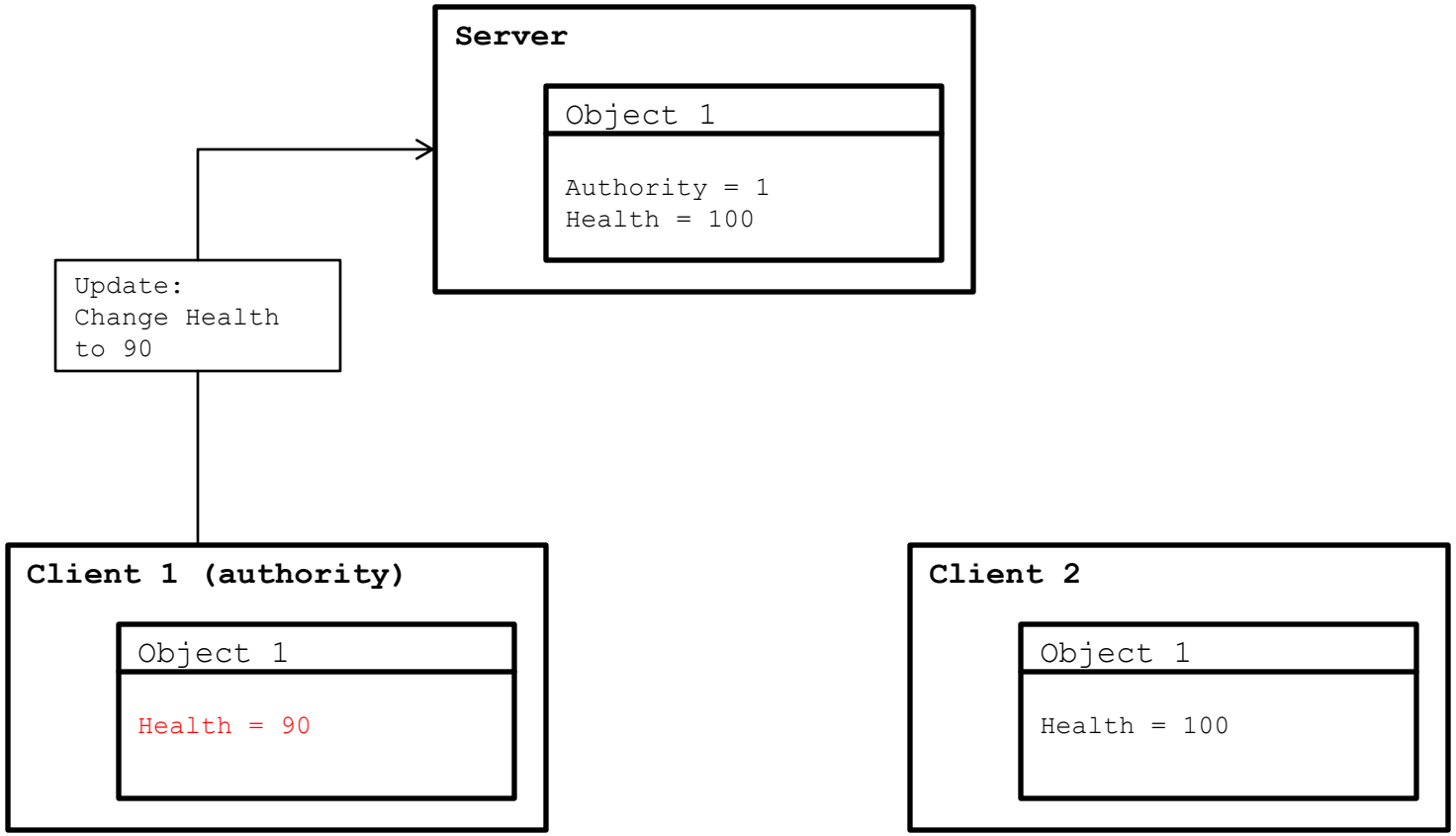
Updates (continued)

- Example: Burning Car
  - Car has object id = 1
  - Starts with 100 health
  - Health is reduced by 10 points per second

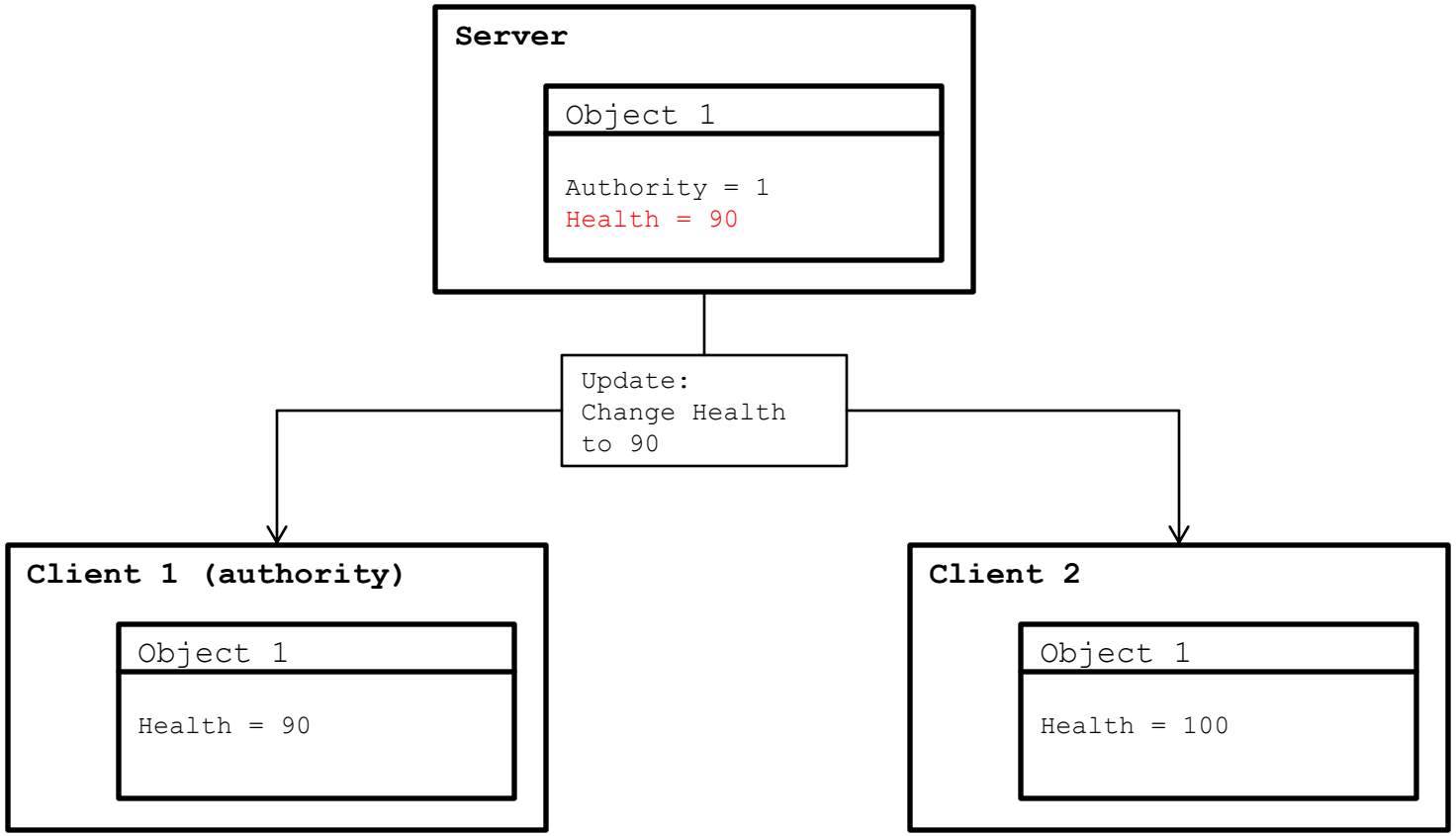
Updates (continued)



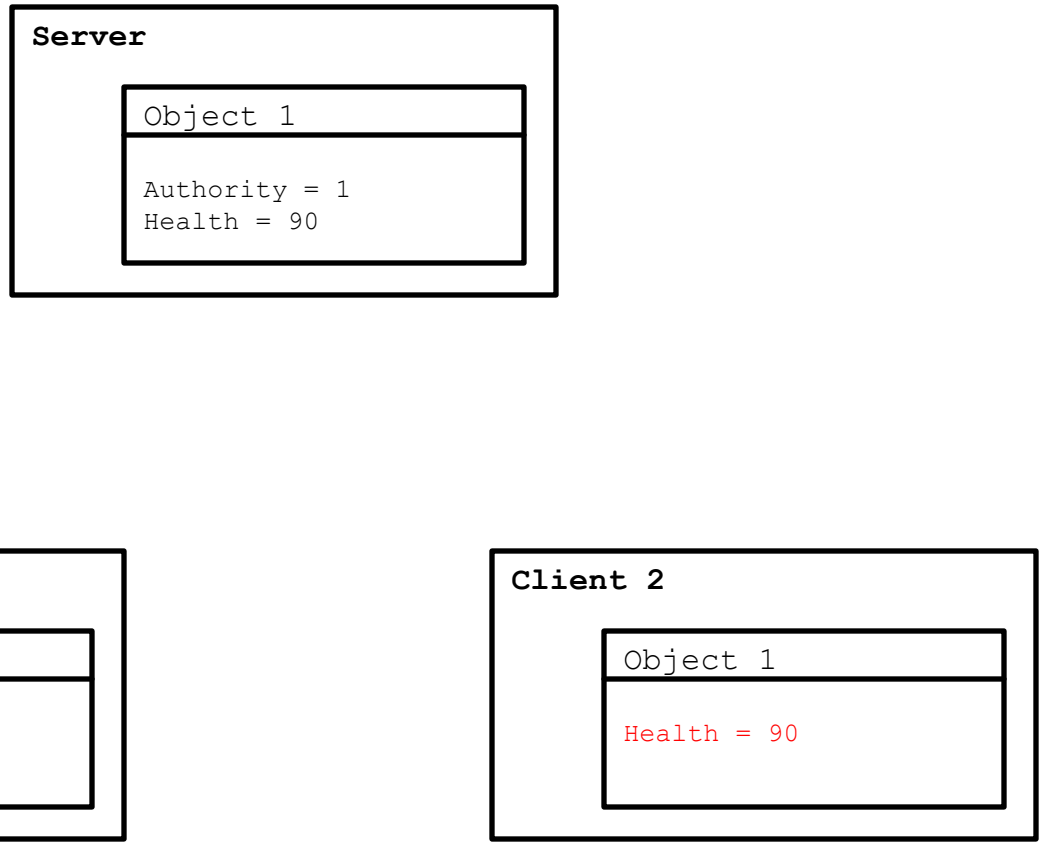
Updates (continued)



Updates (continued)



Updates (continued)



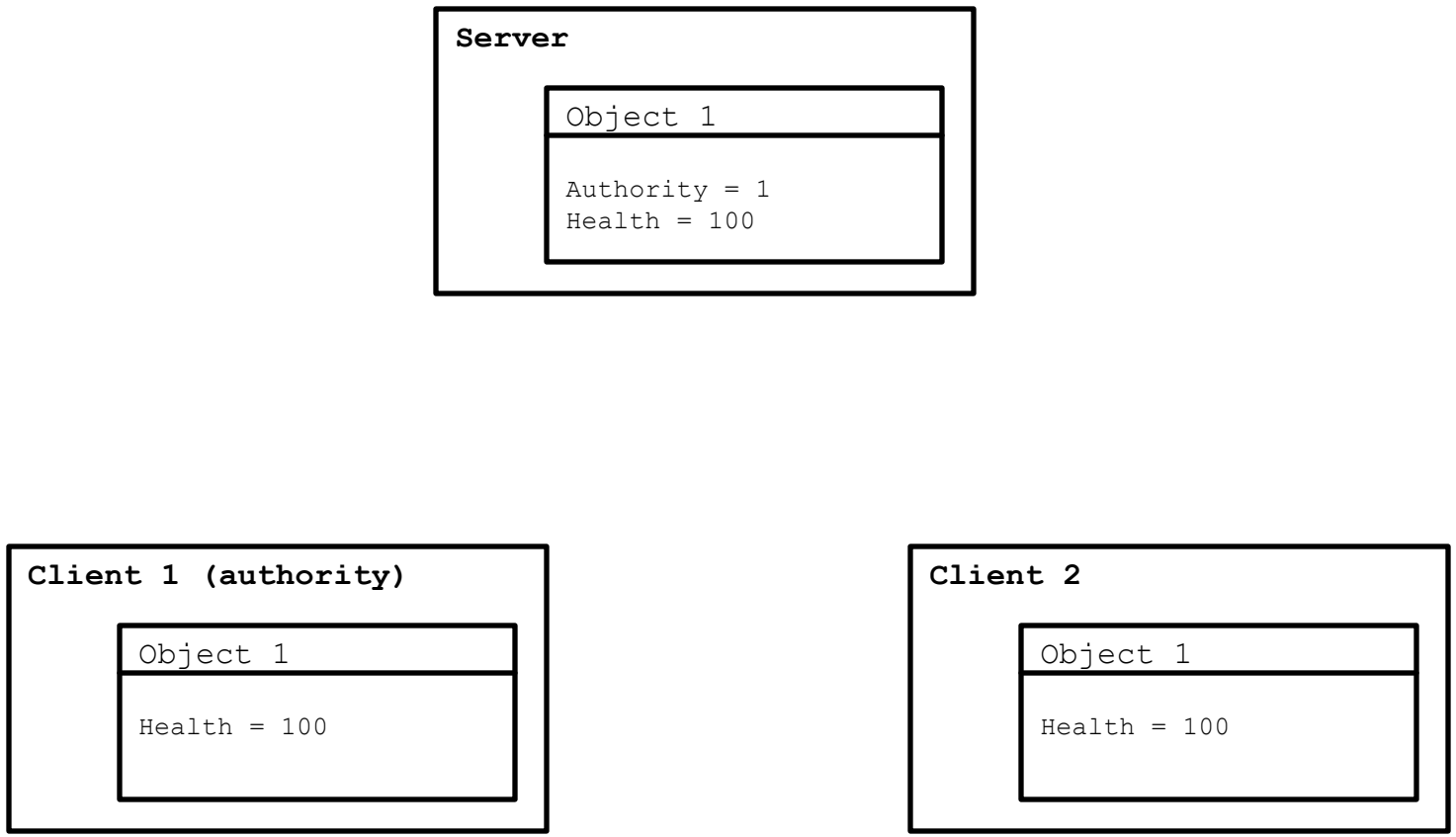
Updates (continued)

- Problem: what happens if the client sending the change loses authority
  - Server will reject health change to 90 since they are no longer the authority
  - But locally, the client has already changed health to 90, leaving them in an inconsistent state with other clients

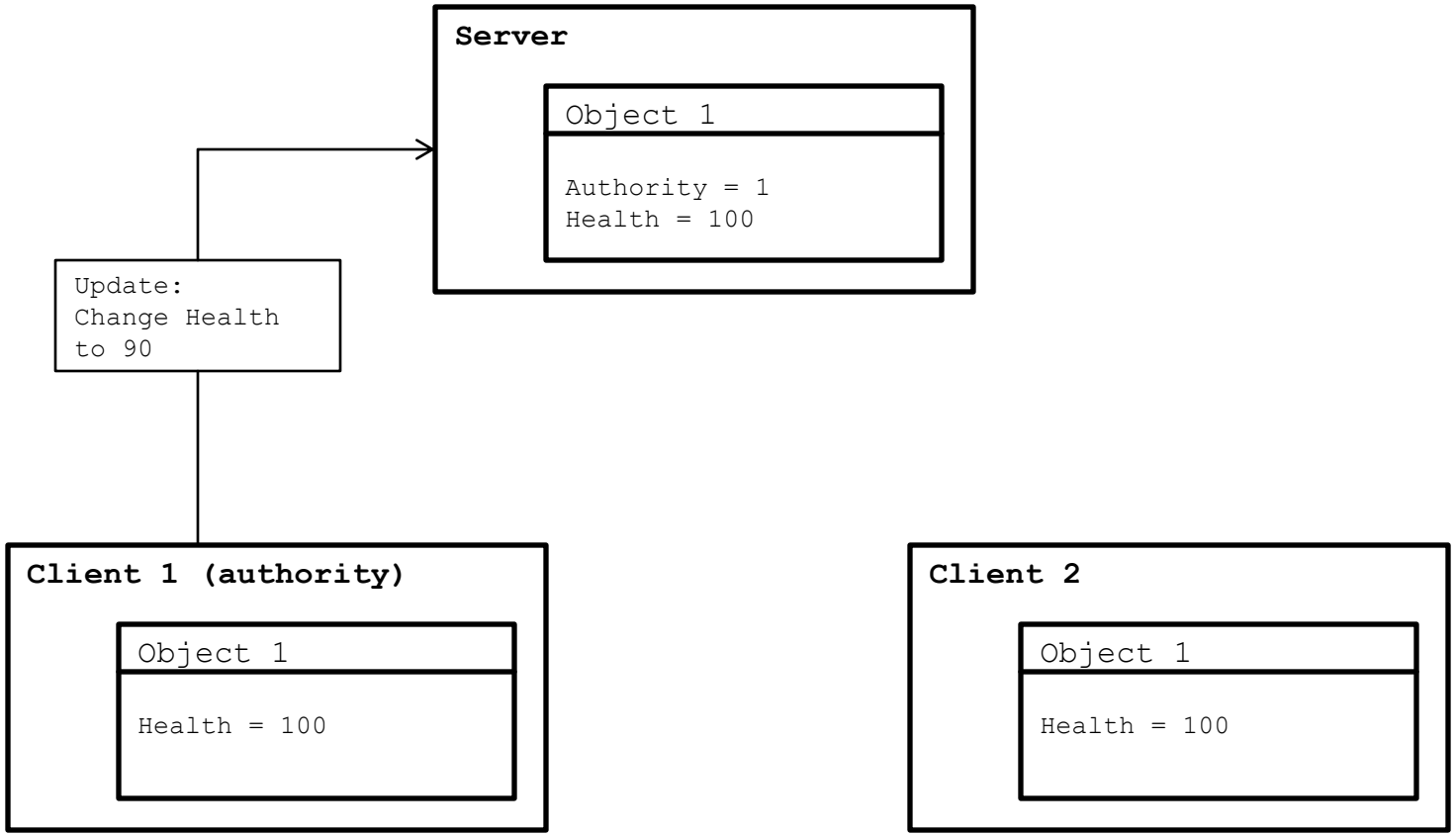
Updates (continued)

- Solution: only send change requests to the server without modifying any local state
  - Authority sees the car has 100 health, knows that it should be 90 and sends change to server
  - Server gets message, changes health to 90, forwards change back to all clients, including authority
  - If authority changes, server will reject change but all clients remain consistent

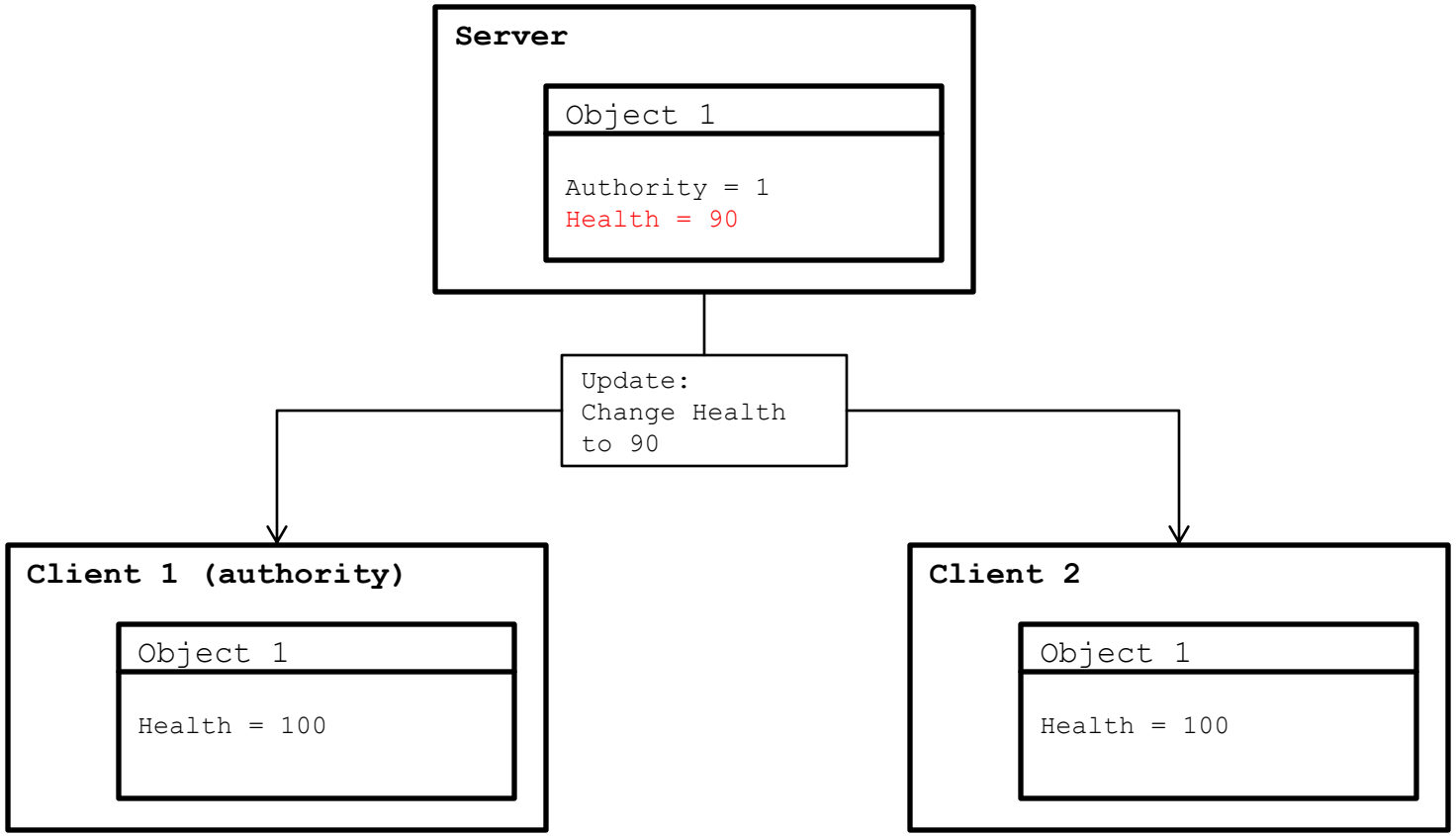
Updates (continued)



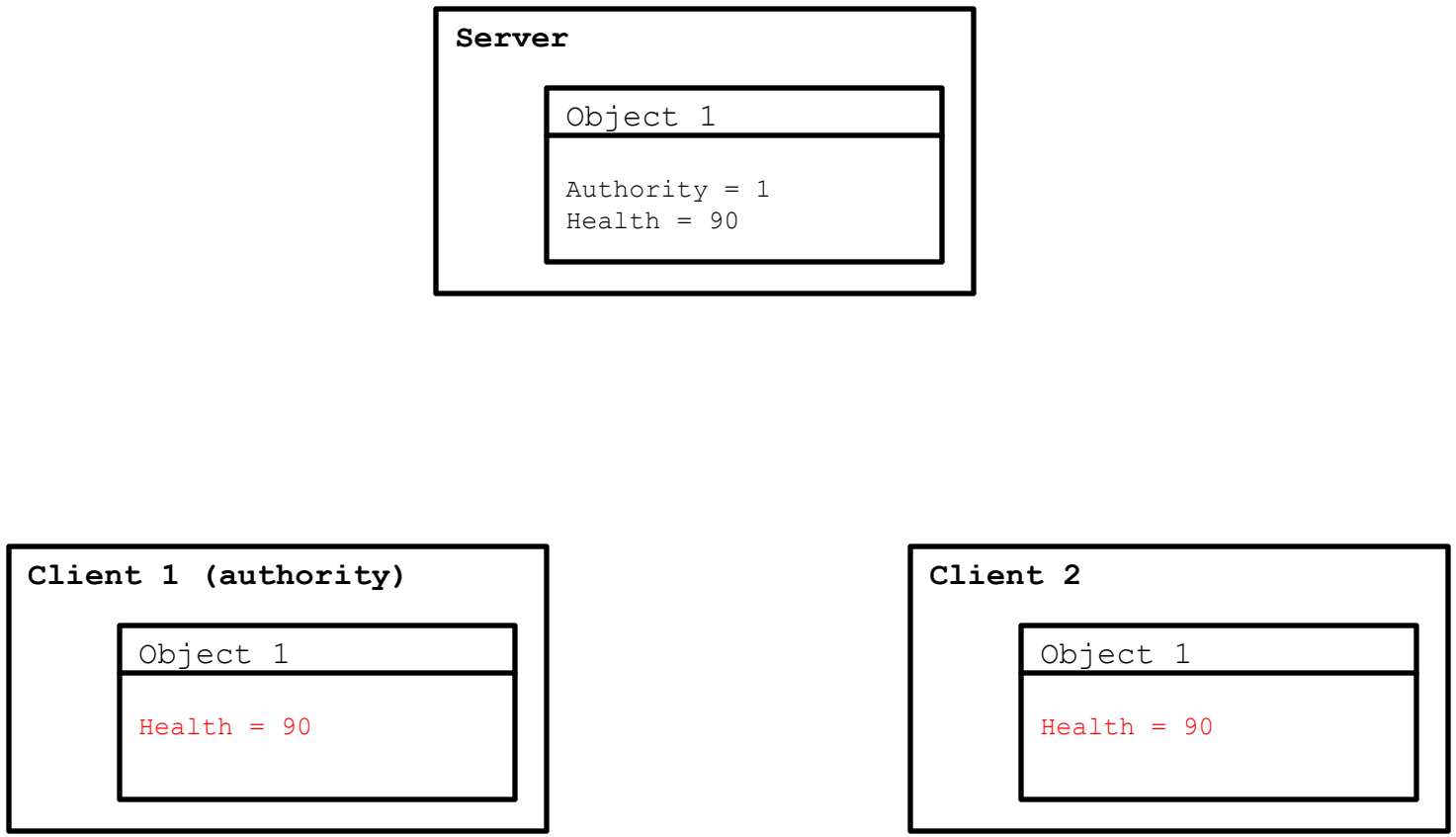
Updates (continued)



Updates (continued)



Updates (continued)



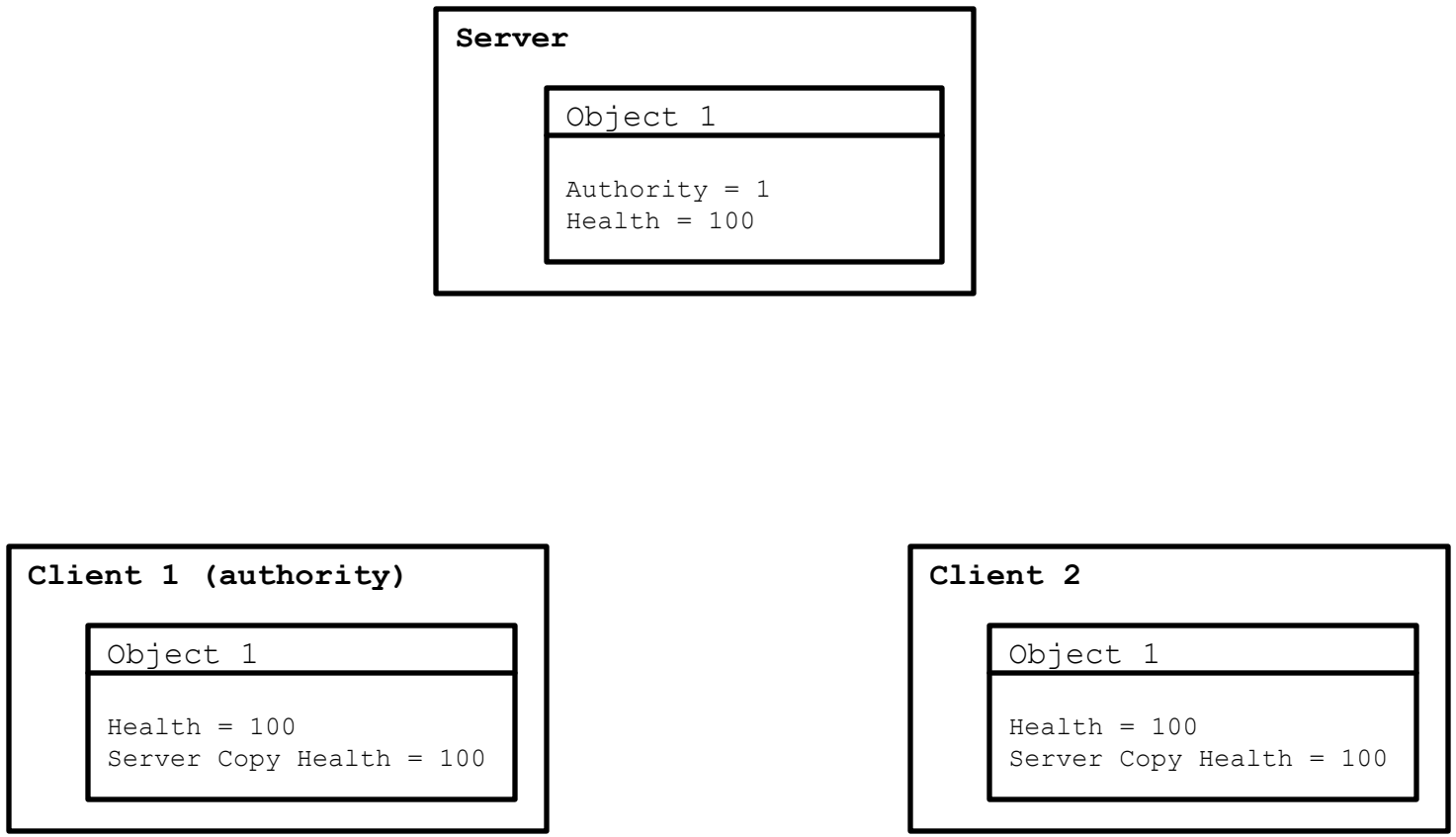
Updates (continued)

- Problem: multiple updates are not queued properly
  - Authority sees car has 100 health, deducts 10 points, sends request to server to change it to 90
  - 1 second later, server has not responded with change yet, so the authority still sees 100 health
  - Server finally sees first update, changes health to 90 and forwards it to clients
  - Server then sees second update, changes health to 90 and forwards it to clients
  - Health gets set to 90 instead of 80

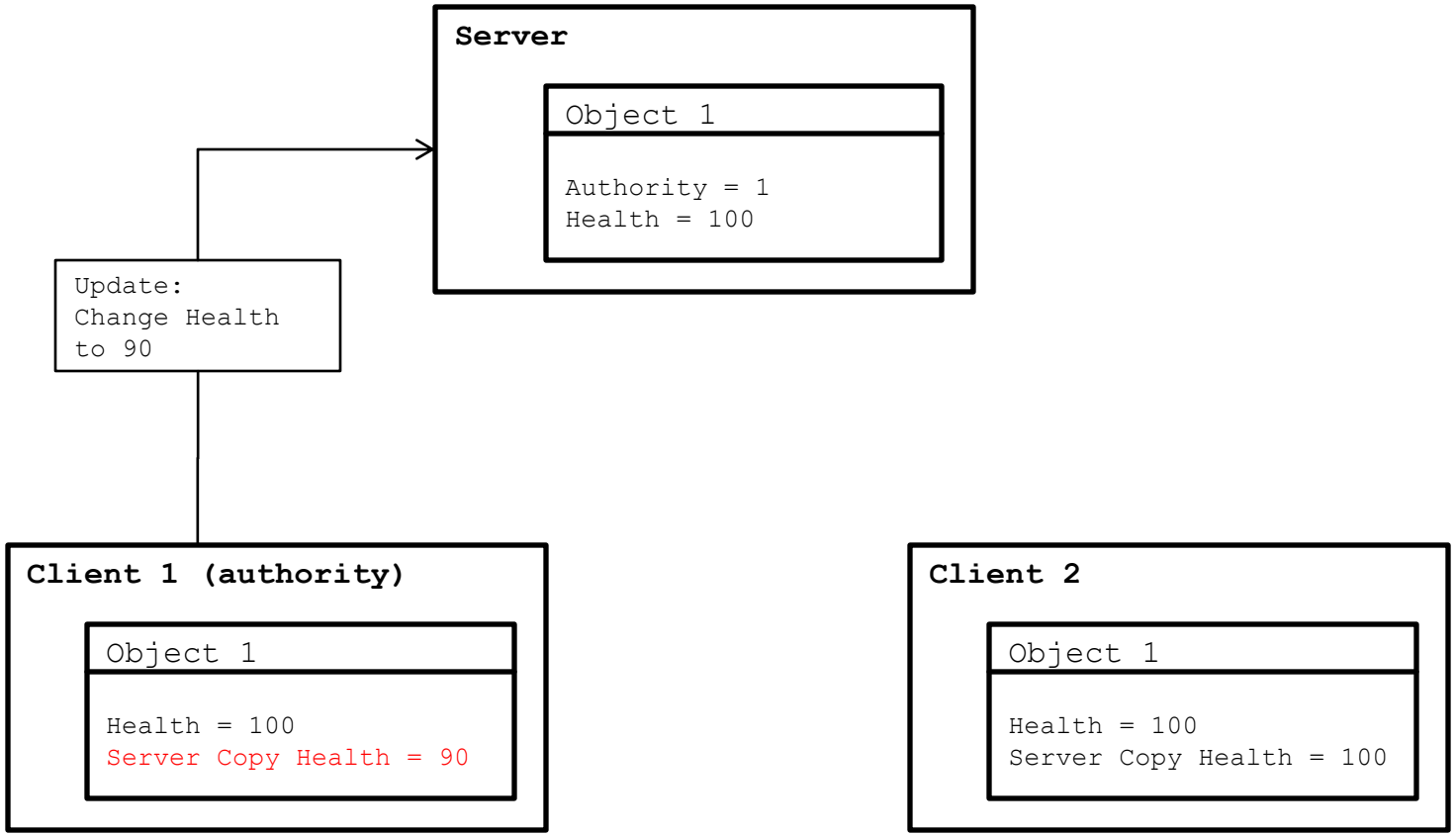
Updates (continued)

- Solution: maintain second copy of each field on the client, known as the server copy
  - Server copy of a field contains the change that was last sent to the server
  - When a non-authority client receives an update, their server copy is overwritten with the change as well
  - Regular game logic still relies on regular copy

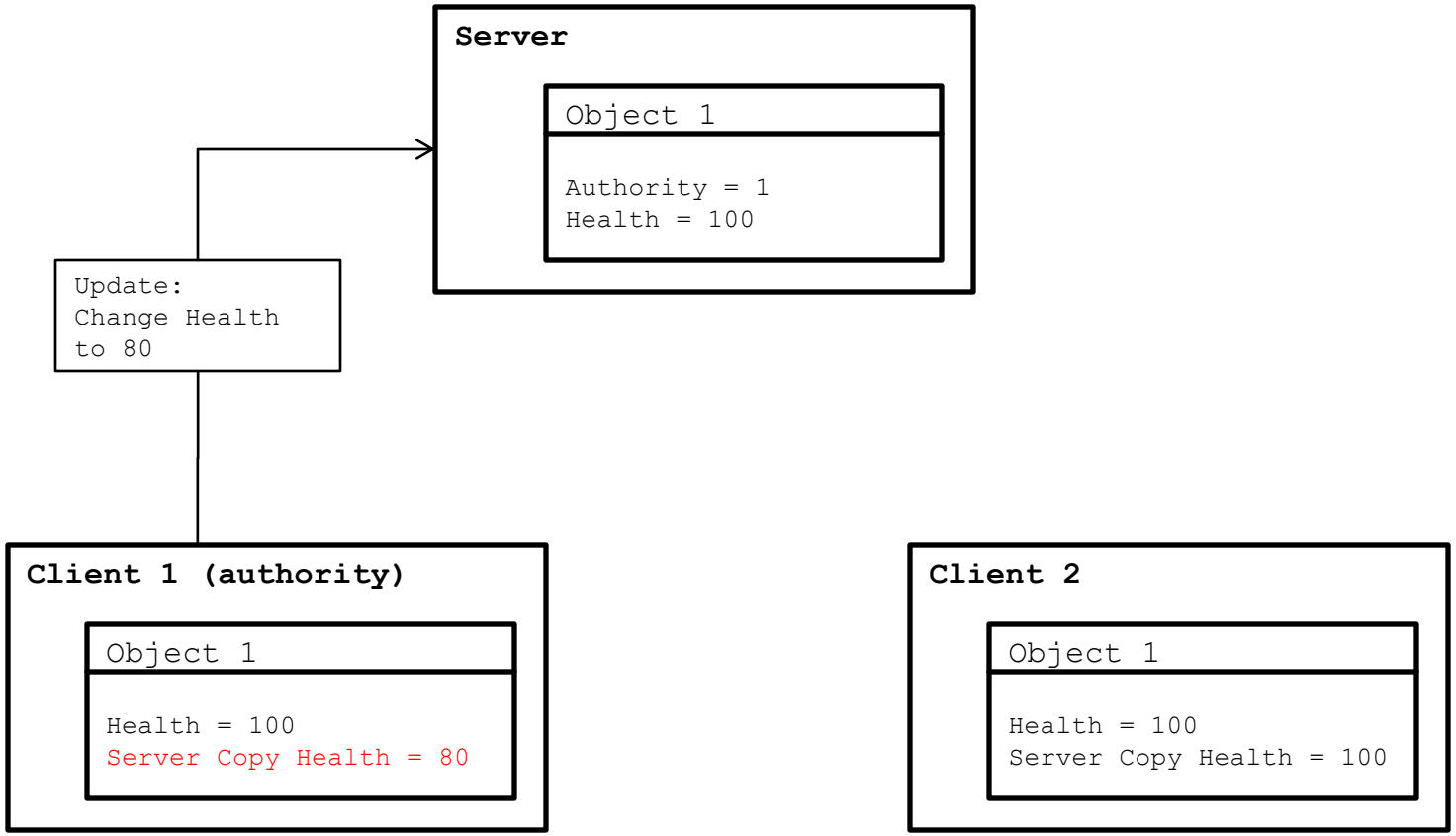
Updates (continued)



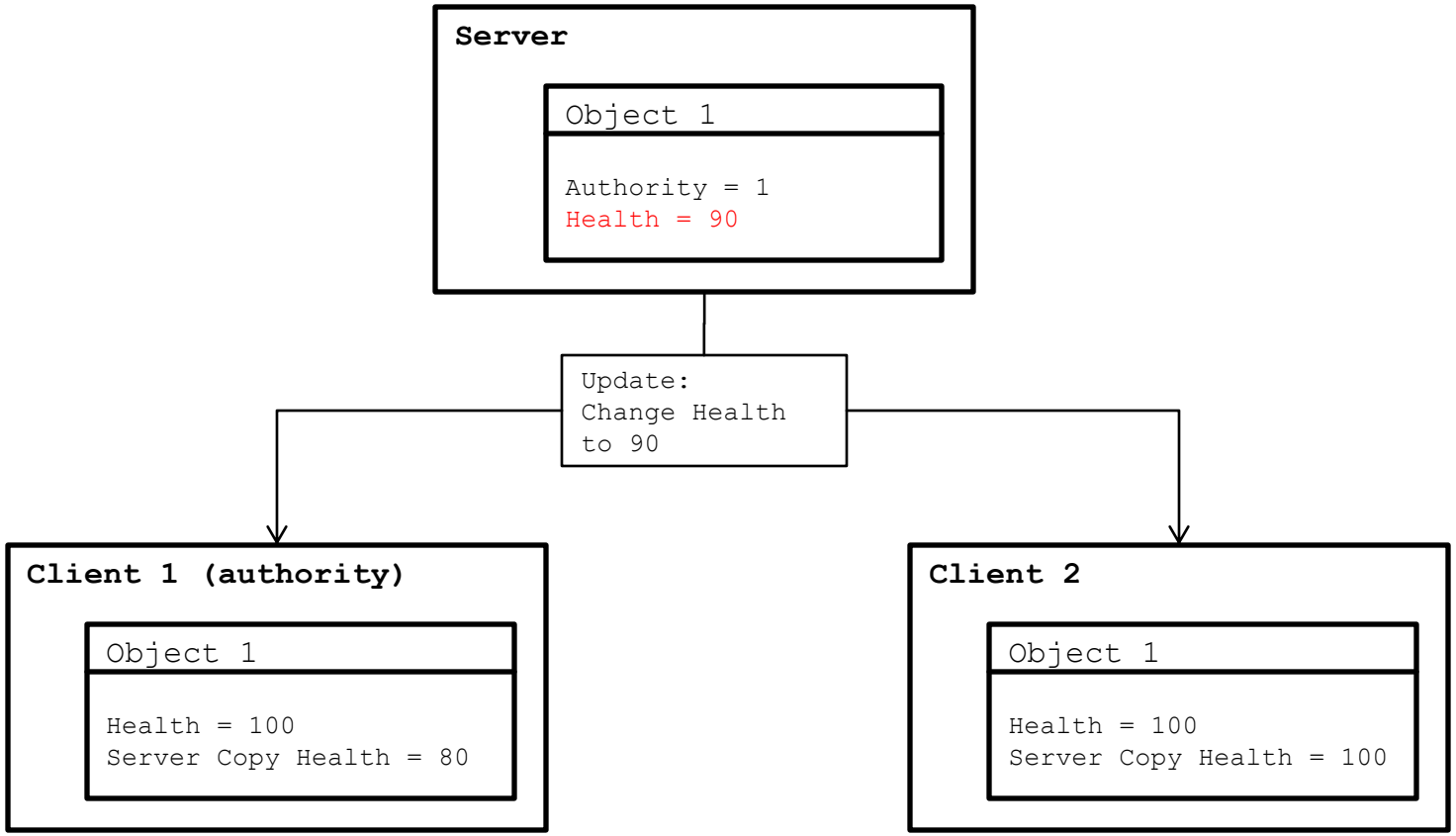
Updates (continued)



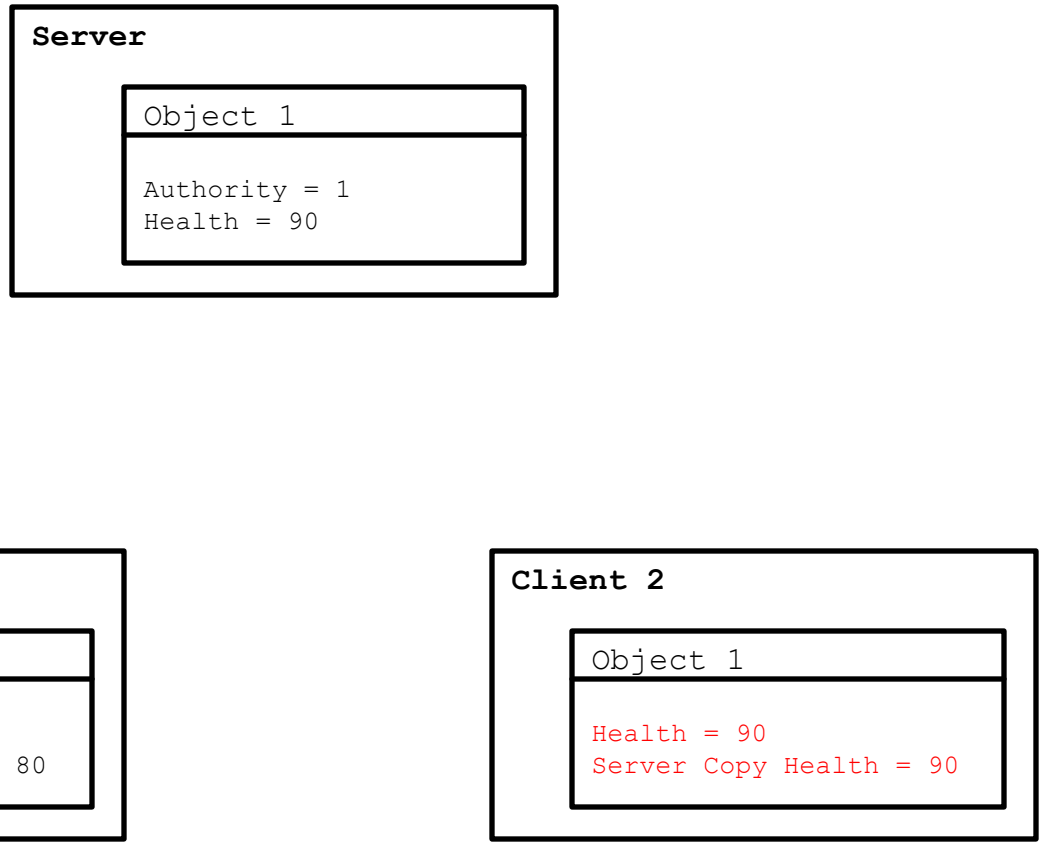
Updates (continued)



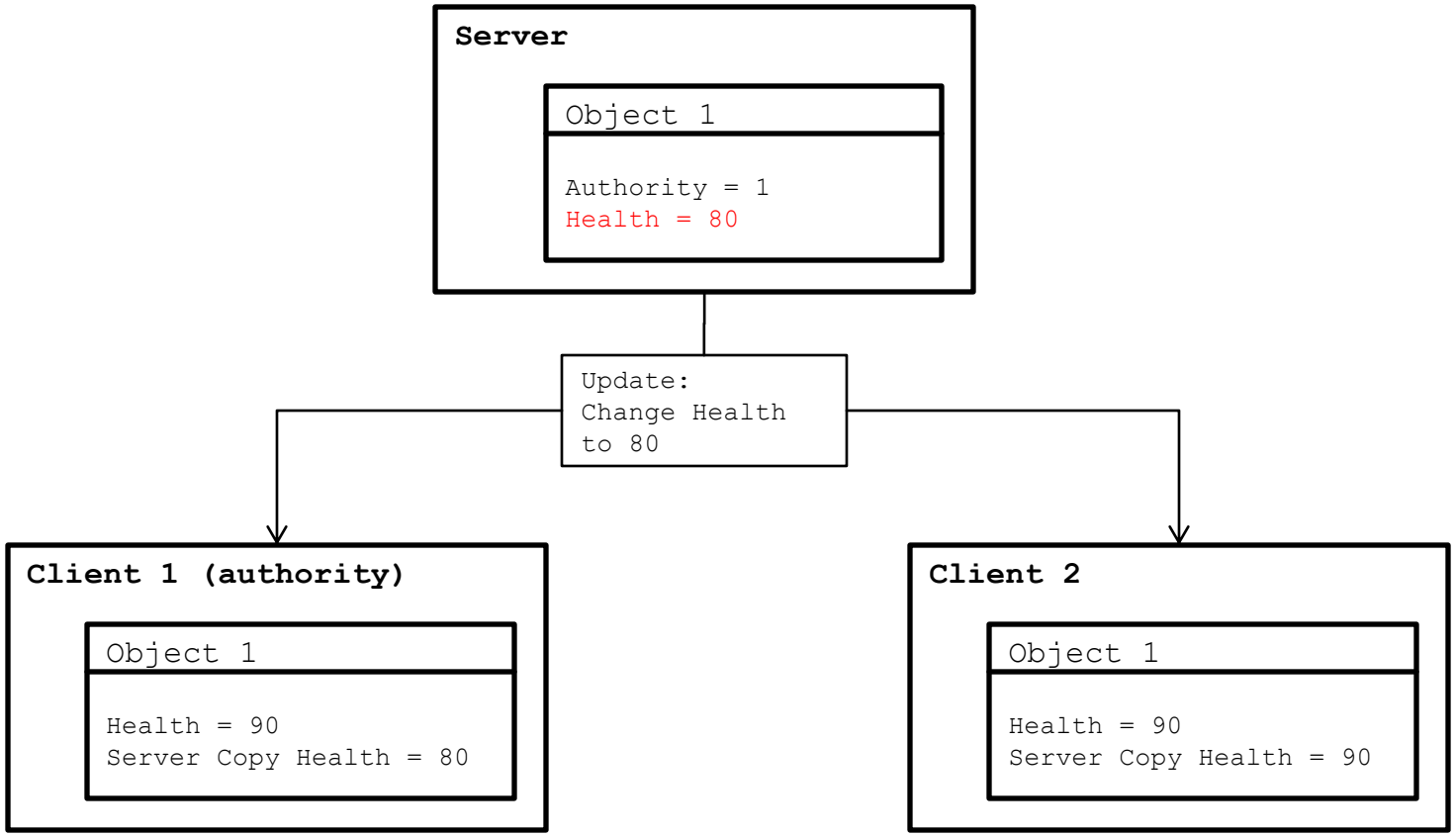
Updates (continued)



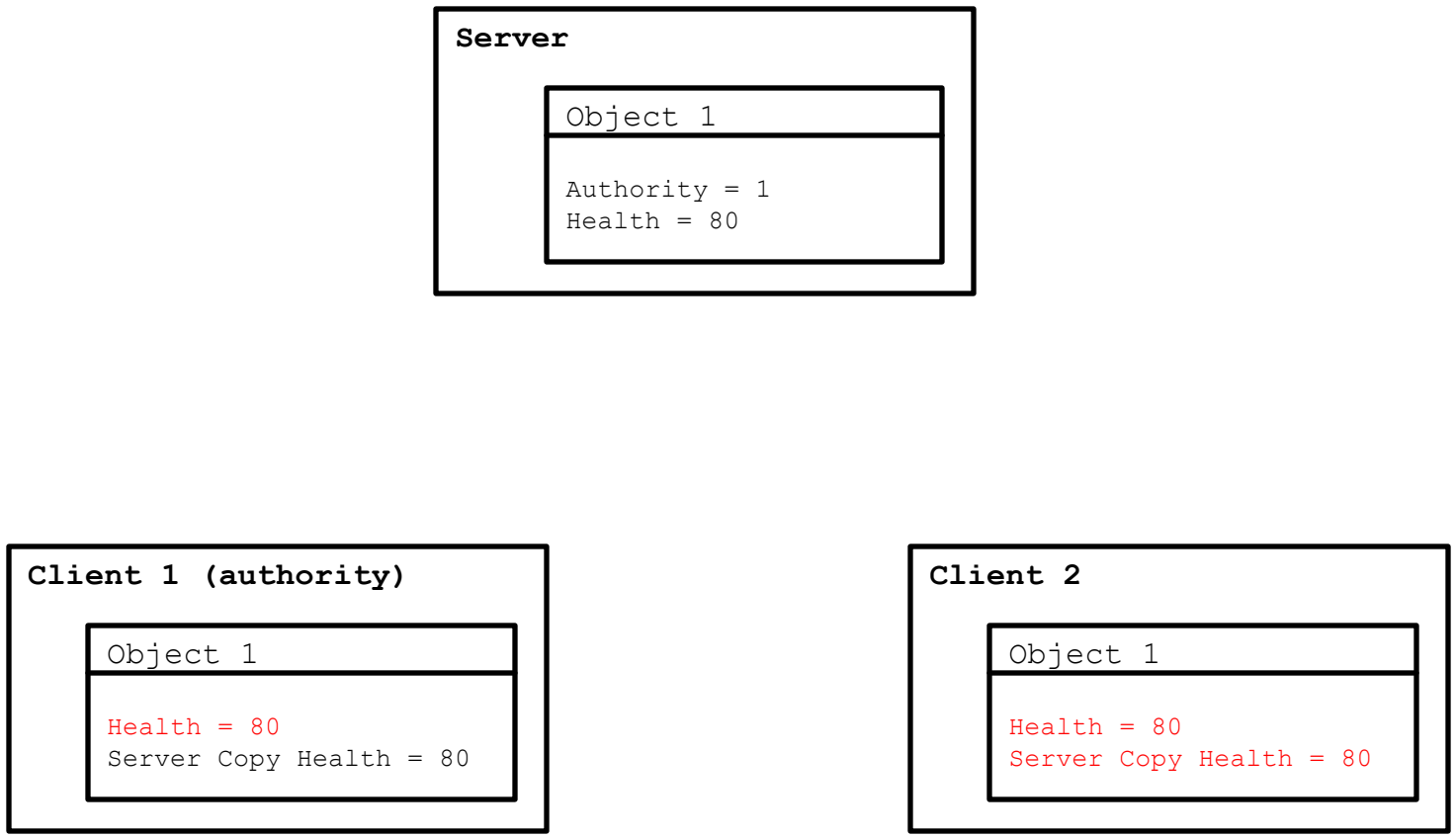
Updates (continued)



Updates (continued)



Updates (continued)



Outline

- Motivation
  - Challenges
  - Solution
- Architecture
  - Objects
  - Authority
- Synchronizing Changes
  - Updates
  - **Events and Responses**
  - Client API

Events and Responses

- A client can send an event on an object to request a change to it
  - Any player can damage a bot – damage messages are events
  - Any player can pick up a weapon – pickup requests are events

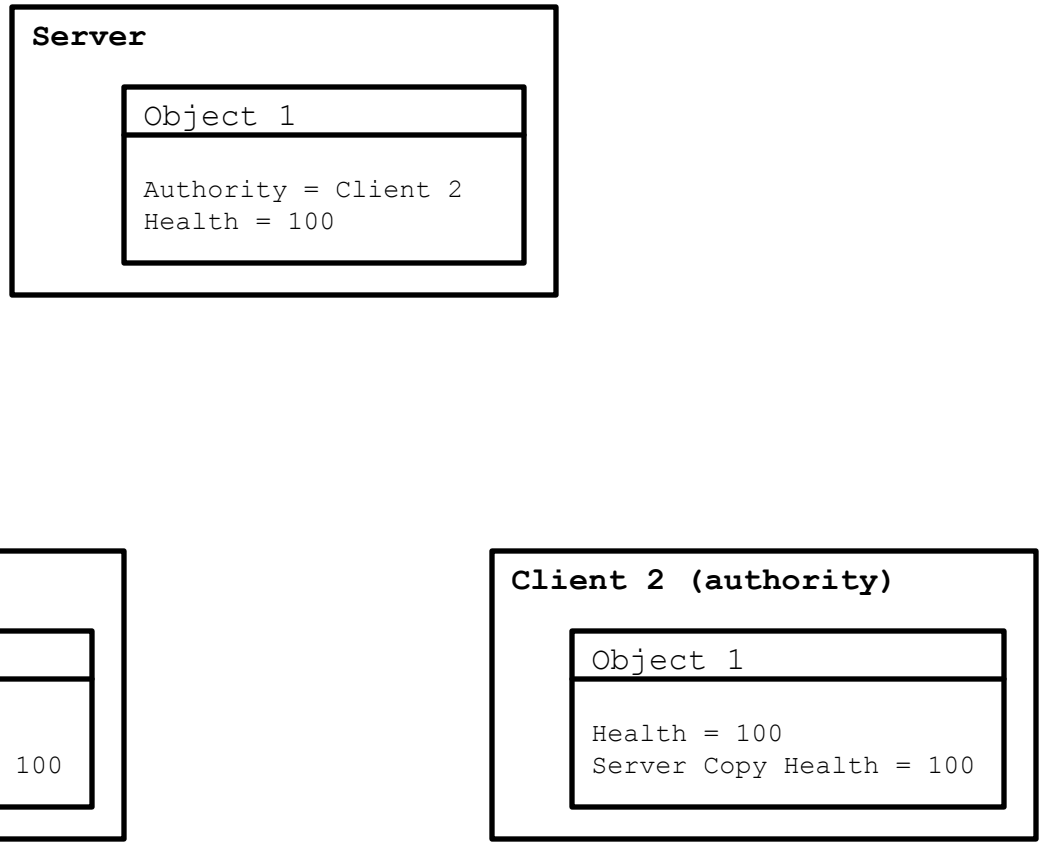
Events and Responses (con't)

- Events routed though server
  - Client 1 sends event to server on object
  - Server determines authority of object
  - Server forwards event to authority
  - Authority handles event and sends a response that contains state changes on the object
  - Responses are similar to updates

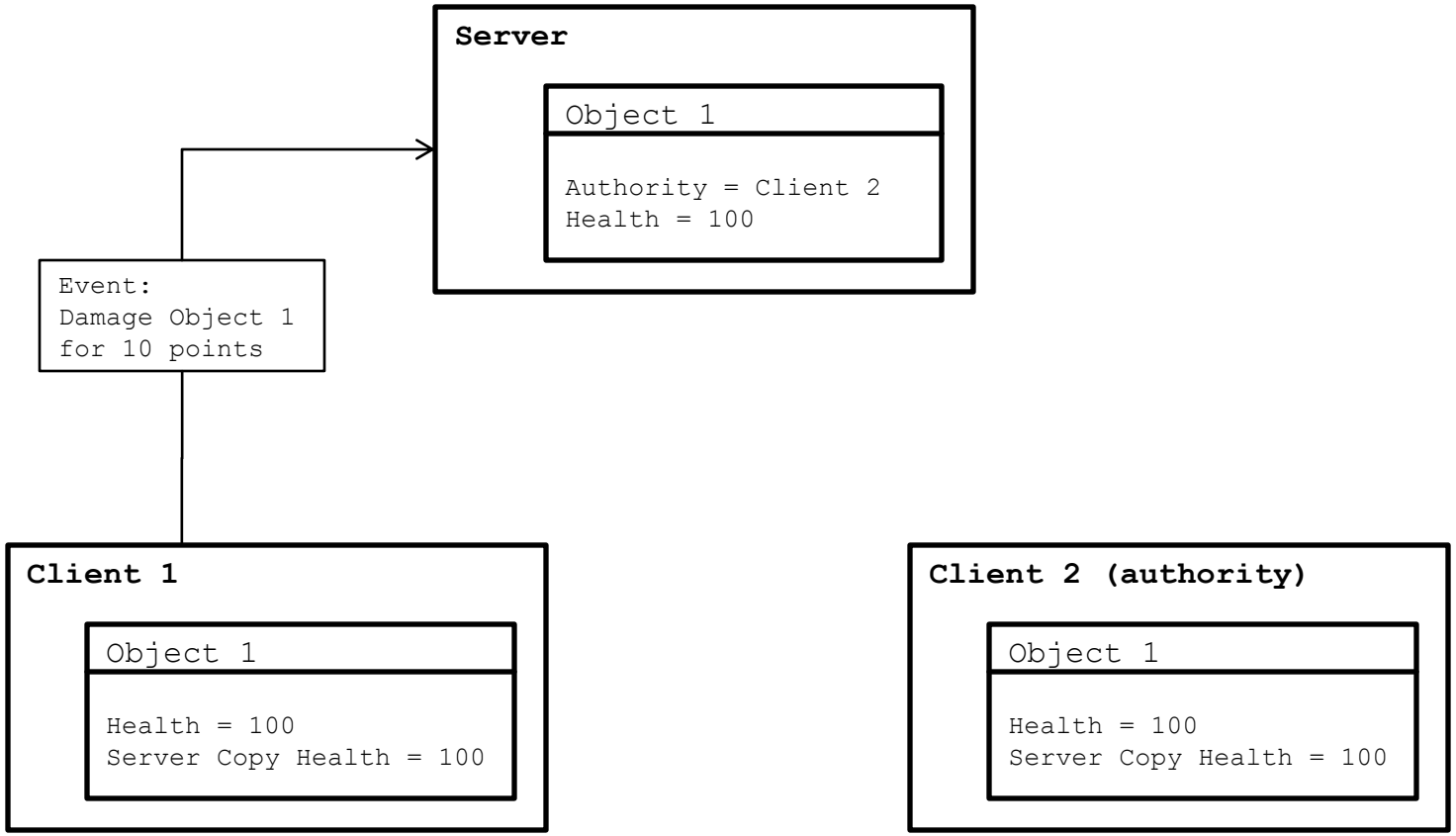
Events and Responses (con't)

- Multiple event types are supported per object
- Each event is defined with an event id
  - An object can have a damage event that represents a request from a client to reduce health
  - Same object can have a repair event that is a request to increase health
  - Damage and repair events would have different ids

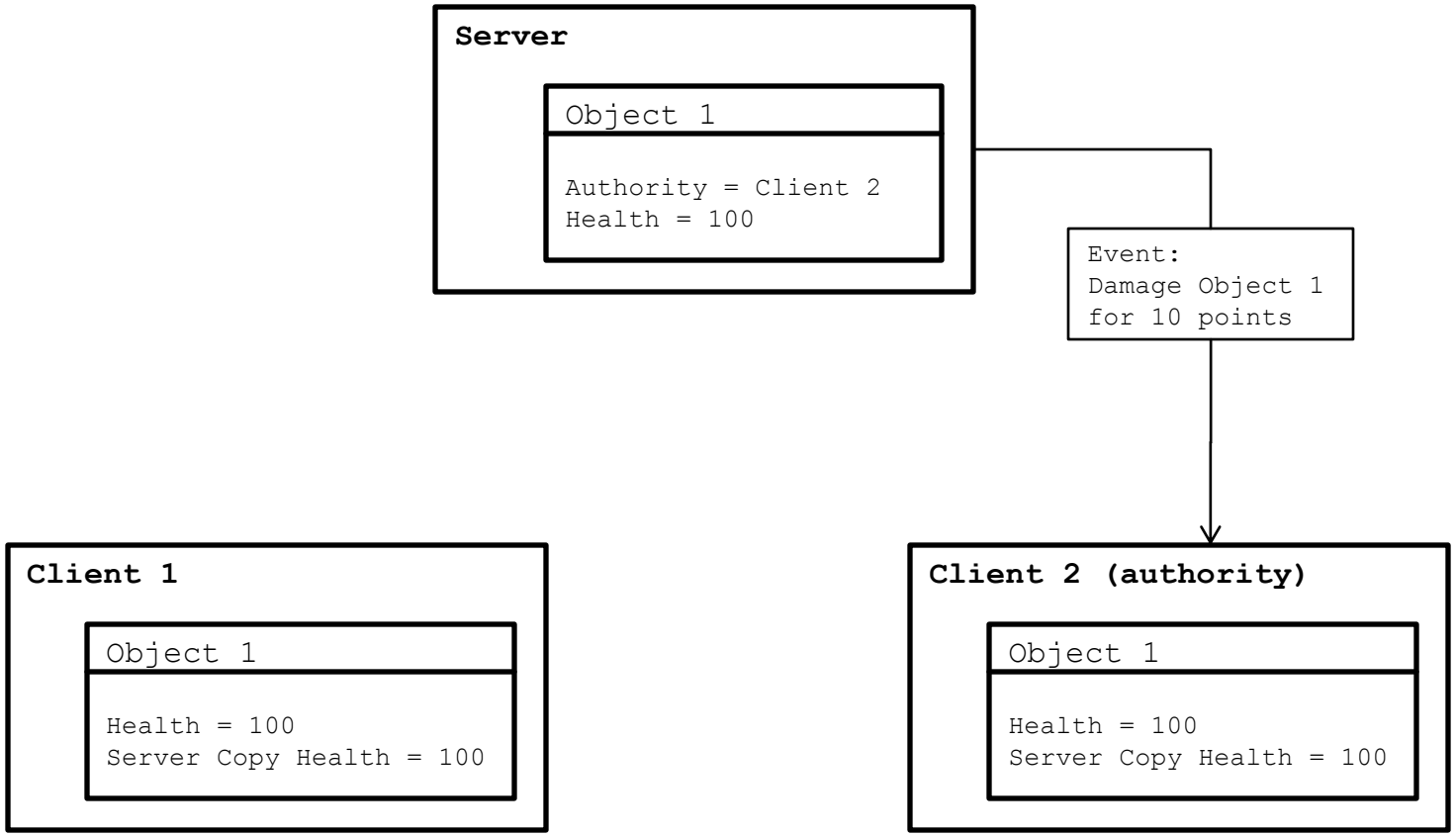
Events and Responses (con't)



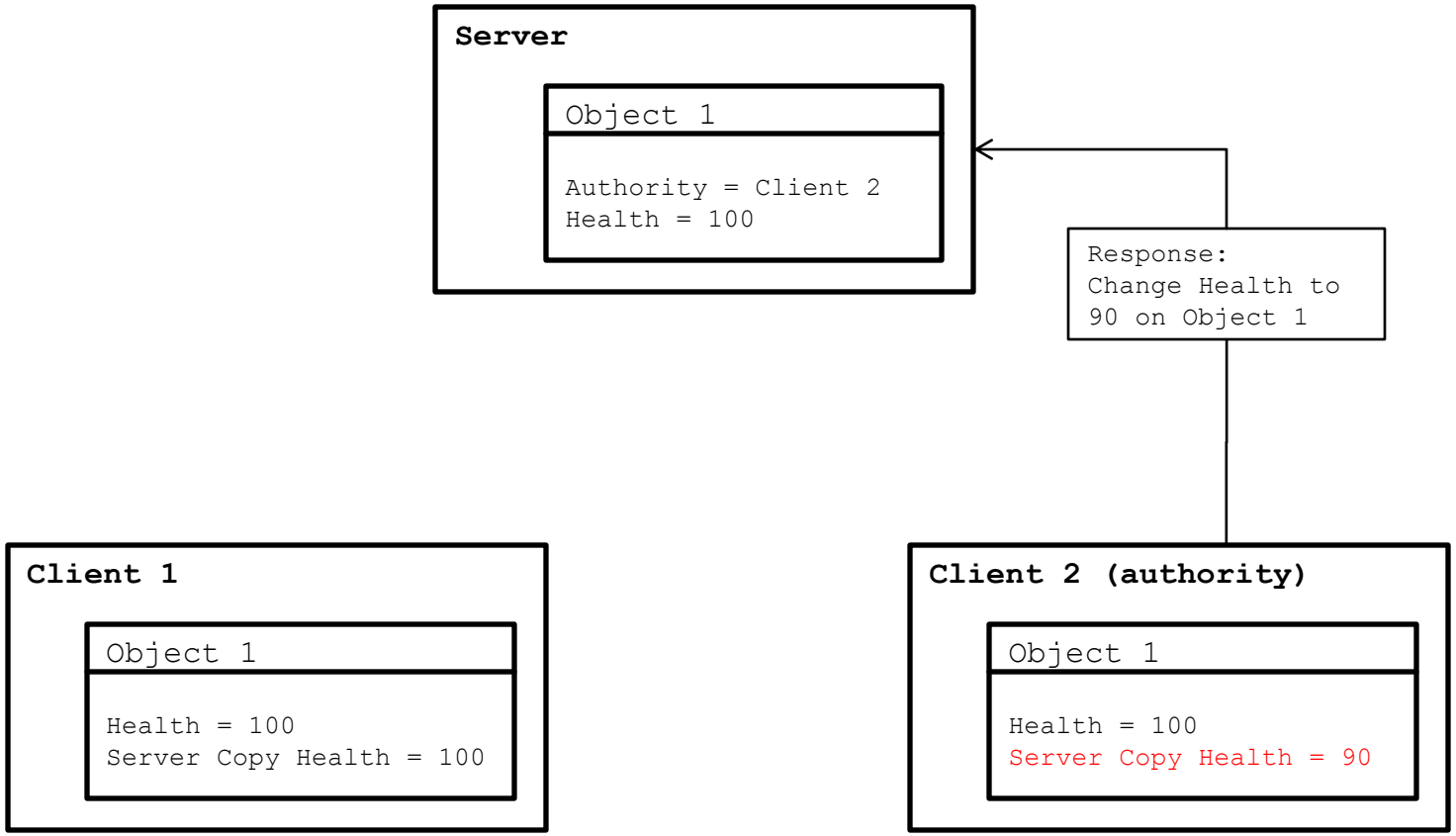
Events and Responses (con't)



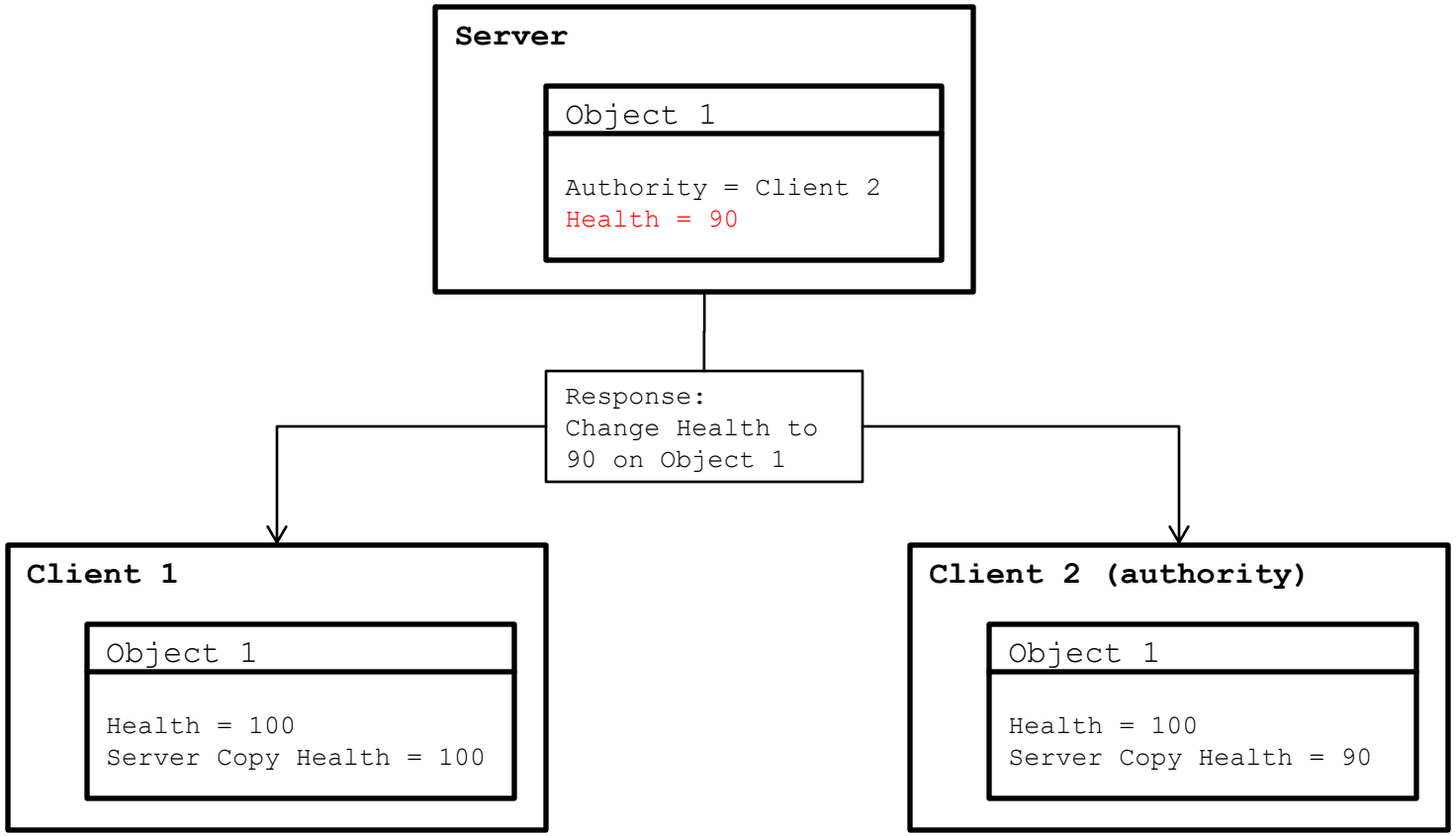
Events and Responses (con't)



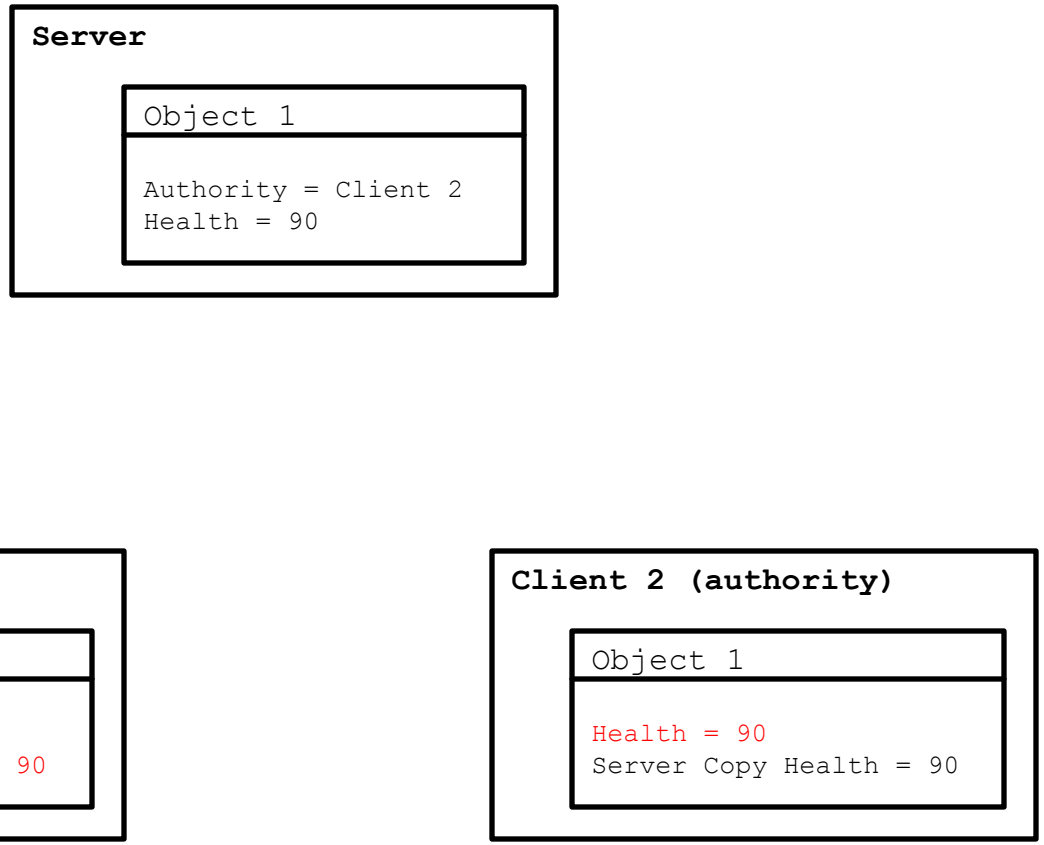
Events and Responses (con't)



Events and Responses (con't)



Events and Responses (con't)



Events and Responses (con't)

- An event is effectively a message that gets the authority of an object to send an update

Outline

- Motivation
  - Challenges
  - Solution
- Architecture
  - Objects
  - Authority
- Synchronizing Changes
  - Updates
  - Events and Responses
  - **Client API**

Client API

- Sync data classes
  - Static object that defines a set of fields and message handlers
  - All instances of the same update class share the same sync data class

Client API (continued)

- Sync managers
  - A sync manager is allocated for each object on the client that needs to be synced
  - Majority of sync host API operates through a sync manager
  - Contains instance-specific data
    - Pointer to object being synced
    - Pointer to sync data class
    - Object id
    - Group id

Client API (continued)

- Example: DamageableCar update class
  - Can take damage, which reduces its health
  - Health also reduces by `10` points per second
  - Has a single synced field, m_health, with field id `0`
  - Has a single event type, damage, with event id `0`

Client API (continued)

- Sample synced update class

```
////////////////////////////////////////////////////////////////////////////////
// DamageableCar - update class for exploding car that can be damaged
////////////////////////////////////////////////////////////////////////////////

// Simple damage event sent for cars
struct DamageEvent : public Sync::SyncEvent
{
  f32 m_damage_amount;
};

class DamageableCar : public GameMobyUpdate
{
public:
  // Synced field ids
  enum
  {
    FIELD_HEALTH = 0
  };

  // Event ids
  enum
  {
    EVENT_DAMAGE = 0
  };
```

Client API (continued)

- Sample synced update class (continued)

```
  virtual void Init();
  virtual void Update();
  virtual void Delete();
  virtual void ProcessDamage( DMG::DamageResult* p_dmg_result );

  // Update handler
  static HandleSyncUpdate( Sync::SyncUpdate* p_update, DamageableCar * p_car );

  // Damage event/response handlers
  static void HandleDamageEvent( DamageEvent* p_event, DamageableCar * p_car );
  static void HandleDamageResponse( Sync::SyncResponse* p_response, DamageableCar * p_car );

protected:
  // Handle to sync manager
  Sync::ManagerHandle m_sync_manager_handle;

  // Synced health field
  f32 m_health;

  // Timer to do damage-over-time
  f32 m_dot_timer;
};
```

Sync data class

```
////////////////////////////////////////////////////////////////////////////
// DamageableCar sync data class - defines what fields on are synced on the
// update class and what message handlers it uses
////////////////////////////////////////////////////////////////////////////

// Starts sync class declaration - sync class is referenced by the name
// passed into the macro
DECLARE_SYNC_CLASS( DamageableCarSyncClass )

  // Field registration - needs name of class fields are on
  BEGIN_FIELDS( DamageableCar )
    // Register fields - requires a field id and member name
    REGISTER_FIELD(DamageableCar::FIELD_HEALTH, m_health );
  END_FIELDS

  // Register update handler - requires message handler
  REGISTER_STATIC_UPDATE_HANDLER( DamageableCar::HandleSyncUpdate )

  // Register event and response handlers - requires an event id and message handler
  REGISTER_STATIC_EVENT_HANDLER( DamageableCar::EVENT_DAMAGE,
    DamageableCar::HandleDamageEvent );
  REGISTER_STATIC_RESPONSE_HANDLER(DamageableCar::EVENT_DAMAGE,
    DamageableCar::HandleDamageResponse );

END_SYNC_CLASS( DamageableCarSyncClass )
```

Client API (continued)

- Sync manager allocation/de-allocation

```
void DamageableCar::Init()
{
  GameMobyUpdate::Init();
  m_health = 100.0f;

  // Allocate sync manager - requires a pointer to a sync data class
  // which can be referenced with the GET_SYNC_CLASS macro
  Sync::SyncManager* p_manager = Sync::CreateManager(
    GET_SYNC_CLASS( DamageableCarSyncClass ) );

  // Pass pointer to our self to sync manager so it can reference synced
  // fields
  p_manager->Init( this );

  // Use server-copy of data when syncing fields
  p_manager->AllocServerCopy();

  // Store handle
  m_sync_manager_handle = p_manager->GetHandle();
}


void DamageableCar::Delete()
{
  // De-allocate sync manager
  Sync::FreeManager( m_sync_manager_handle );

  GameMobyUdpate::Delete();
}
```

Client API (continued)

- Update example

```
void DamageableCar::Update()
{
  GameMobyUpdate::Update();

  // Get sync manager
  Sync::SyncManager* p_manager = Sync::GetSyncManager( m_sync_manager_handle );

  // Check if we're the authority of the object
  if( p_manager && p_manager->IsAuthority() )
  {
    // Only the authority runs the below code

    // Reduce health 10 points per second
    if( TIME::DecTimer( &m_dot_timer ) )
    {
      // Determine the amount of health we last updated the server with
      f32 server_copy_health;
      p_manager->GetServerField( DamageableCar::FIELD_HEALTH, &server_copy_health );

      // Still have health, send update
      if( server_copy_health > 0.0f )
      {
         Sync::SyncUpdate sync_update;

         // Update server copy of health
         server_copy_health = Maxf( server_copy_health - 10.0f, 0.0f );
         p_manager->SetServerField( DamageableCar::FIELD_HEALTH, &server_copy_health );

         // Mark health as a field to sync
         sync_update.SendField( DamageableCar::FIELD_HEALTH );

         // Send update
         p_manager->Update( &sync_update );
      }

      // Reset dot timer
      m_dot_timer = 1.0f;
    }
  }
}
```

Client API (continued)

```
////////////////////////////////////////////////////////////////////////////
// This function gets called on all clients after the authority sends an
// update
////////////////////////////////////////////////////////////////////////////
void DamageableCar::HandleSyncUpdate( Sync::SyncUpdate* p_update, DamageableCar * p_car )
{
  // Trigger some effects if we've taken too much damage
  if( p_update->FieldIsModified( DamageableCar::FIELD_HEALTH ) )
  {
    if( p_car->m_health <= 50.0f && !p_car->HasTrackedEffects( EventType::SMOKING ) )
    {
      p_car->TriggerTrackedEffectEvent( EventType::SMOKING );
    }
  }
}
```

Client API (continued)

- Event-response example

```
////////////////////////////////////////////////////////////////////////////////
// This function gets called when any local client damages the car - sends off
// an event to the authority with the amount of damage we want to do
////////////////////////////////////////////////////////////////////////////////
void DamageableCar::ProcessDamage( DMG::DamageResult* p_dmg_result )
{
  // Get sync manager
  Sync::SyncManager* p_manager = Sync::GetSyncManager( m_sync_manager_handle );
  if( p_manager )
  {
    // Build sync event to pass damage amount
    DamageEvent dmg_event;
    dmg_event.m_damage_amount = p_dmg_result->m_damage.m_amount;

    // Send event - the first argument is the event id, which will determine
    // which event handler gets called with this event on the authority
    // client
    p_manager->Event( DamageableCar::EVENT_DAMAGE, &dmg_event );
  }
}
```

Client API (continued)

```
////////////////////////////////////////////////////////////////////////////////
// This function gets called on the authority client when any client sends
// a damage event on this damageable car
////////////////////////////////////////////////////////////////////////////////
void DamageableCar::HandleDamageEvent( DamageEvent* p_damage_event, DamageableCar* p_car )
{
  // Get sync manager
  Sync::SyncManager* p_manager = Sync::GetSyncManager( p_barrel->m_sync_manager_handle );
  if( p_manager )
  {
    // Build sync response
    Sync::SyncResponse sync_response;

    // Get current health
    f32 server_copy_health;
    p_manager->GetServerField( DamageableCar::FIELD_HEALTH, & server_copy_health );

    // Check if we're still alive
    if( server_copy_health > 0.0f )
    {
       // Update server copy of health
       server_copy_health = Maxf( server_copy_health - p_damage_event->m_damage_amount, 0.0f );
       p_manager->SetServerField( DamageableCar::FIELD_HEALTH, &server_copy_health );

       // Mark health as a field to sync
       sync_update.SendField( DamageableCar::FIELD_HEALTH );
    }

    // Send response - uses same event id we used when sending original event
    // always gets sent so server can resolve events
    p_manager->Response( DamageableCar::EVENT_DAMAGE, &sync_response );
  }
}
```

Client API (continued)

```
////////////////////////////////////////////////////////////////////////////////
// This function gets called on all clients after the authority has handled
// a damage event and decremented the health of the car
////////////////////////////////////////////////////////////////////////////////
void DamageableCar::HandleDamageResponse( Sync::SyncResponse* p_response, DamageableCar * p_car )
{
  // Trigger some effects if we've taken too much damage
  if( p_response->FieldIsModified( DamageableCar::FIELD_HEALTH ) )
  {
    if( p_car->m_health <= 50.0f && !p_car->HasTrackedEffects( EventType::SMOKING ) )
    {
      p_car->TriggerTrackedEffectEvent( EventType::SMOKING );
    }

    // Trigger a damaged event
    p_car->TriggerEffectEvent( EventType::DAMAGED );
  }
}
```