Sync: why, what, and how

Lev Novik

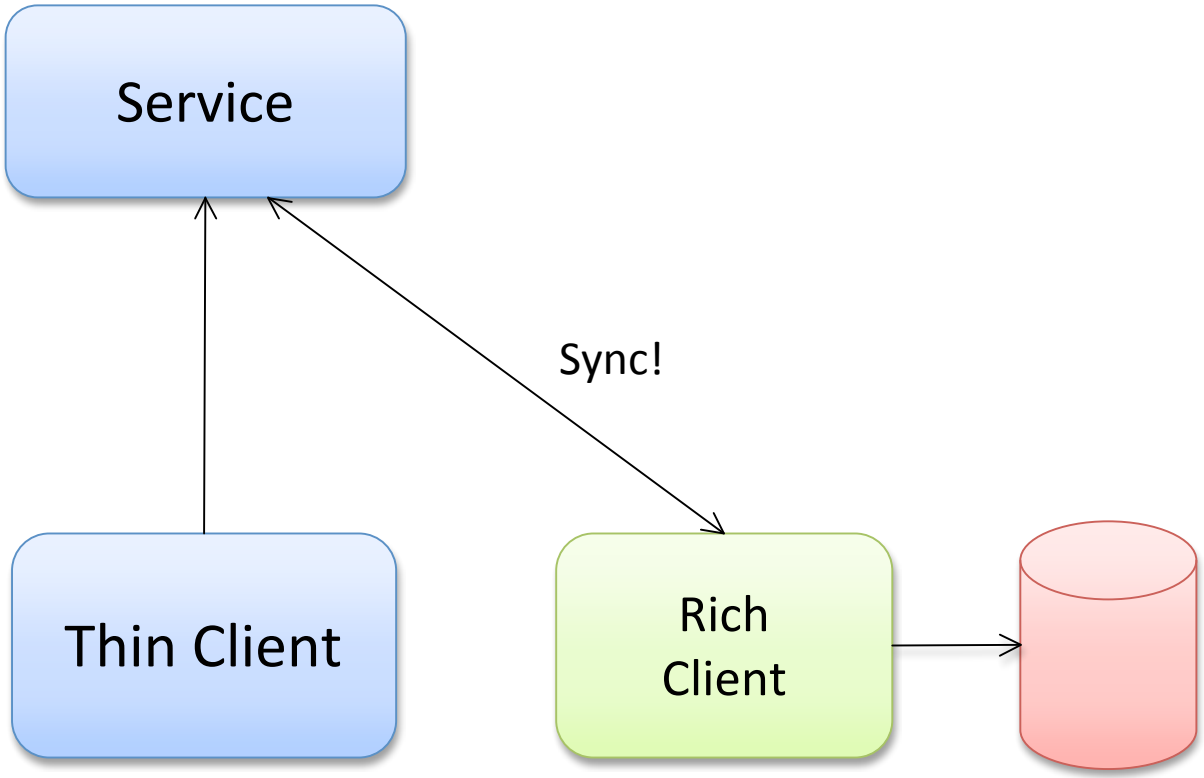
Architect

Microsoft

Agenda

- **Why Sync?**
  - Great taste, less filling
- **Harder than it looks**
  - Problems and Approaches
- **Microsoft Sync Framework**
  - What we do, what we don't
  - How to use us

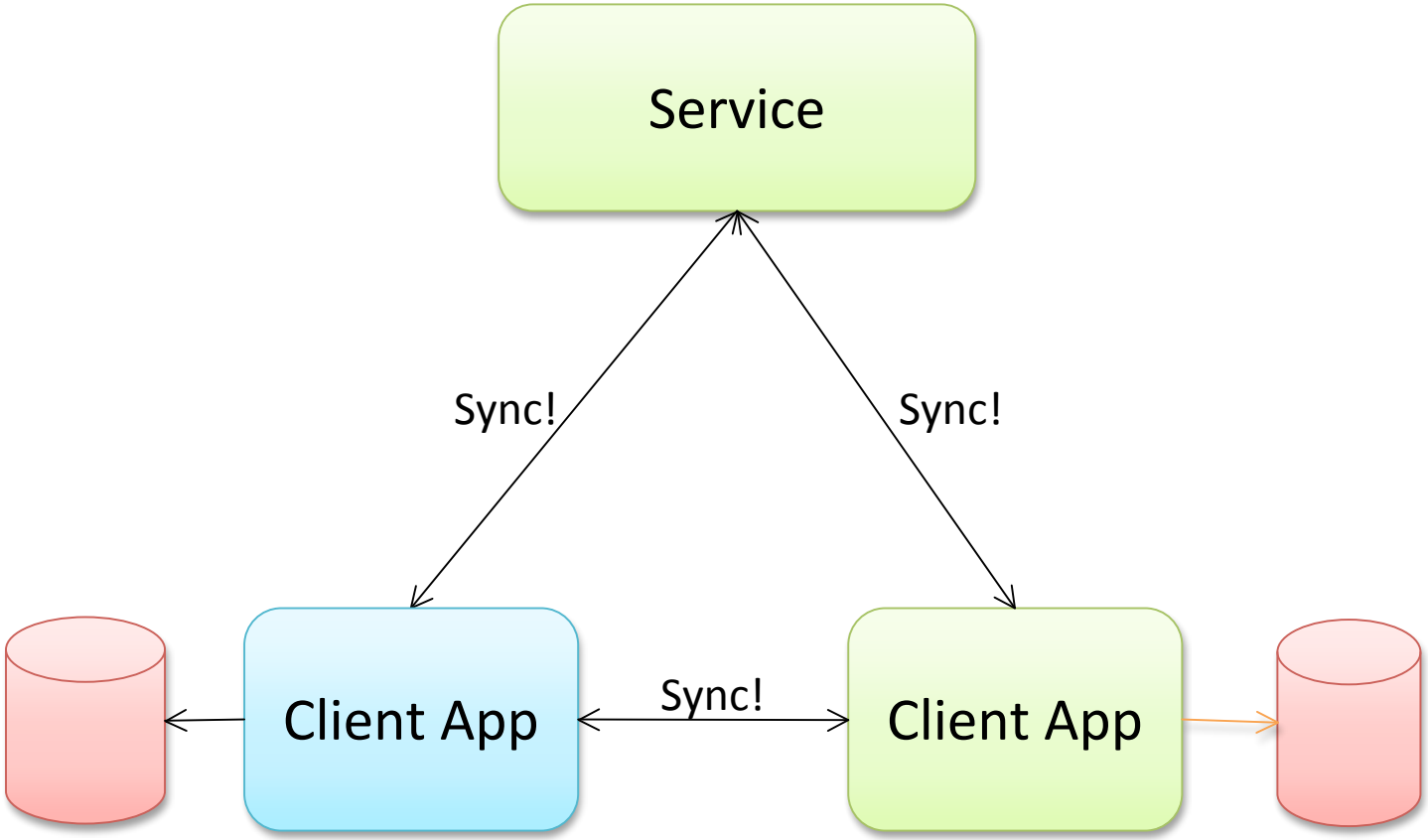
Why Sync?



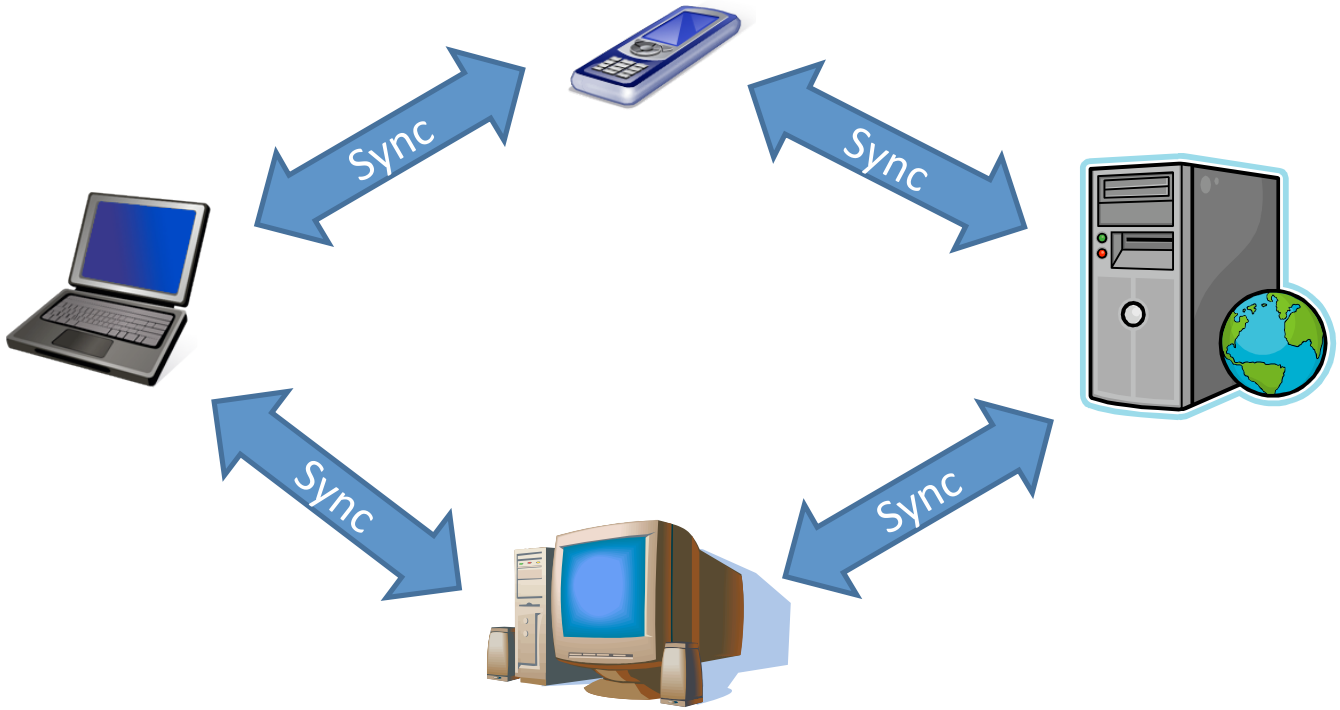
Why Sync?

- Facilitates rich clients
  - Faster **response**, richer UX
- Facilitates better **availability**
  - Sync results in replication that translates to better availability
- Facilitates improved **network utilization**
  - Offline clients generally transmit a lot less
- Facilitates lower **COGS** on server/cloud
  - Significantly reduced READ loads and relative fewer CUD loads

Why Sync?



Sync is ~~hard~~ not easy

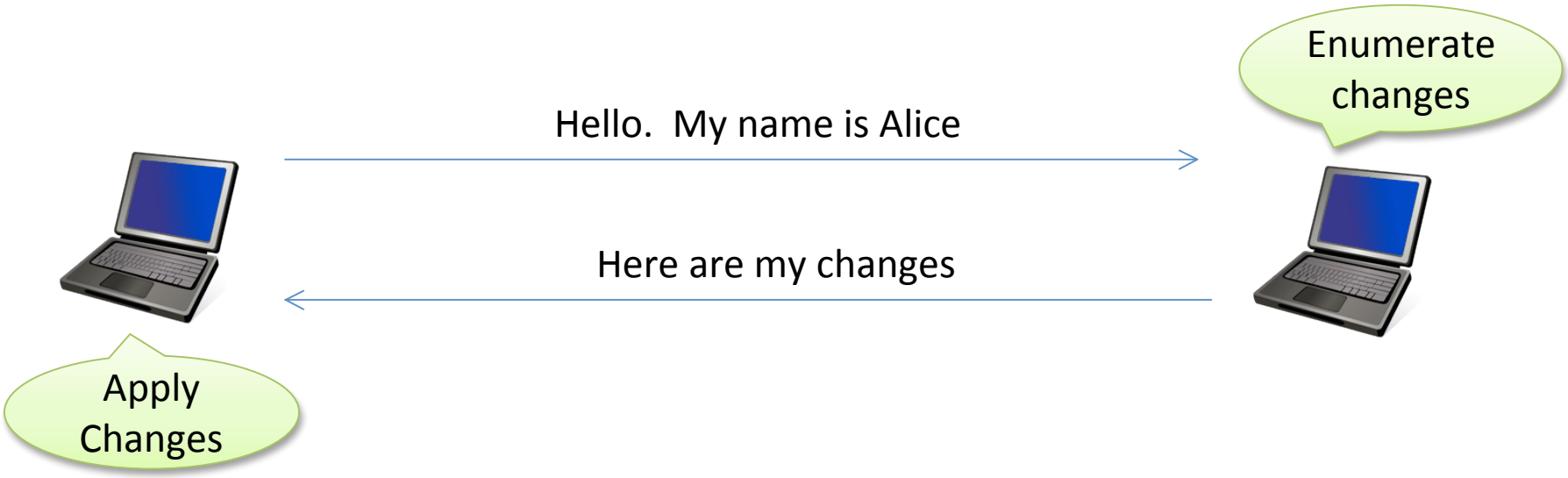


Warning: professional driver on a closed course

Always use protective eyewear

Adult supervision required

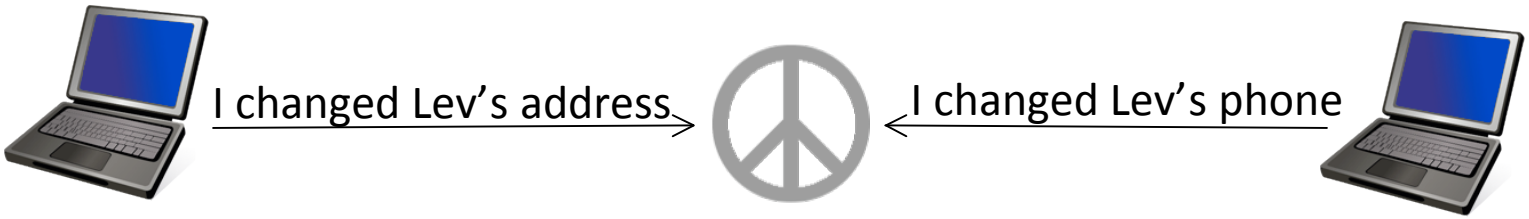
High-level schematics



Enumerating changes

- Needs a reliable watermark
  - To avoid over- and under-enumeration
- Needs to support stores that don't have one
  - Including legacy stores
- Needs to be efficient in large stores
  - Friendly to indexing

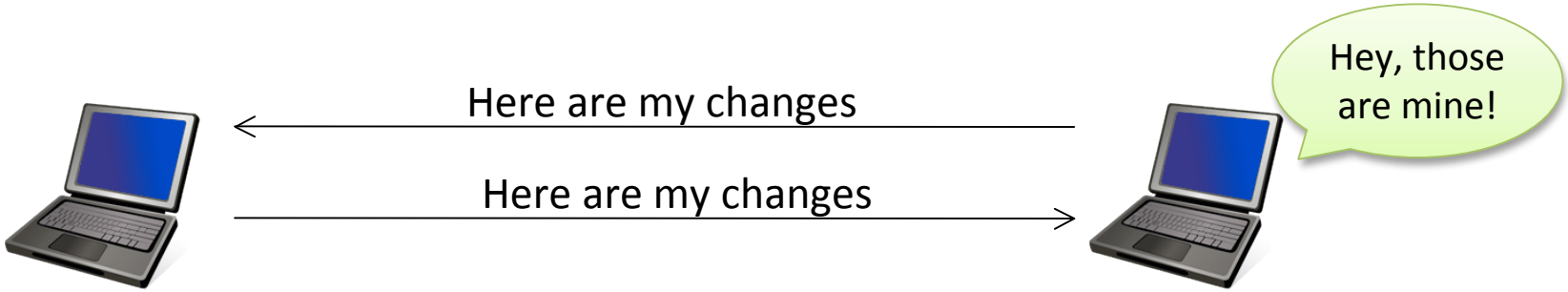
Granularity of Changes



Problems:

- Tracking at item level:
  - Too many conflicts
  - Too much data sent
- Tracking below item level:
  - Too much metadata
  - Lack of consistency

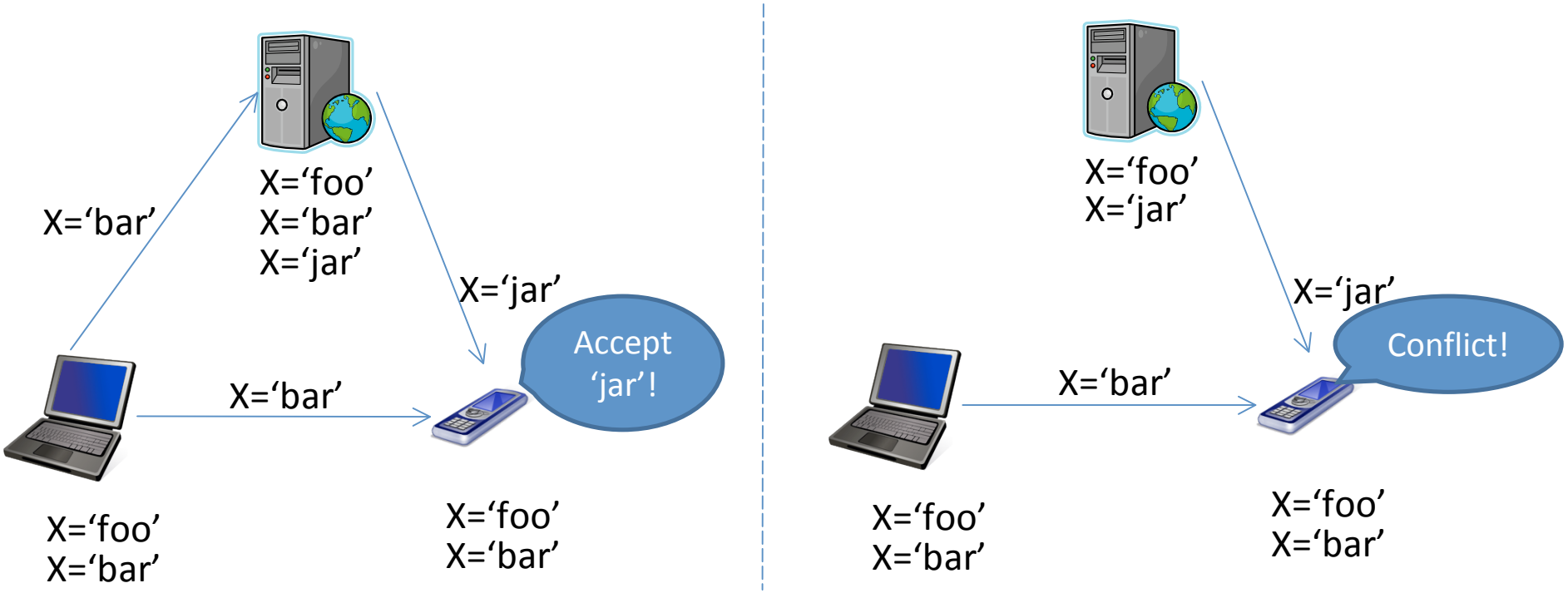
Change (non-)Reflection



With a simple timestamp,

- If you grab timestamp before sync:
  - *change reflection*
  - thus inefficiency and false conflicts
- If you grab timestamp after sync:
  - *missing changes* or
  - *locked stores*

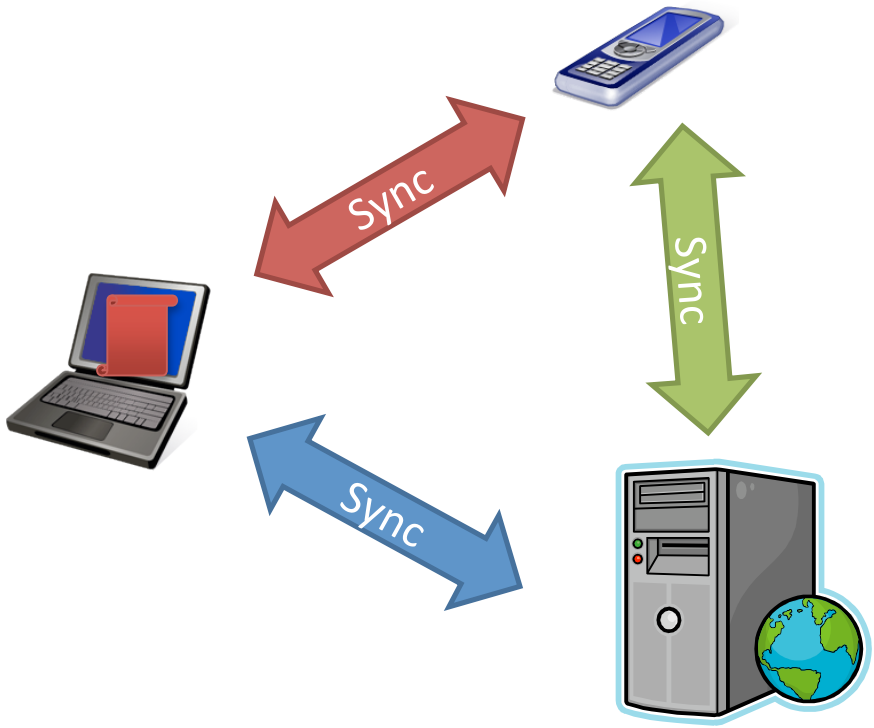
Conflicts



Problems:

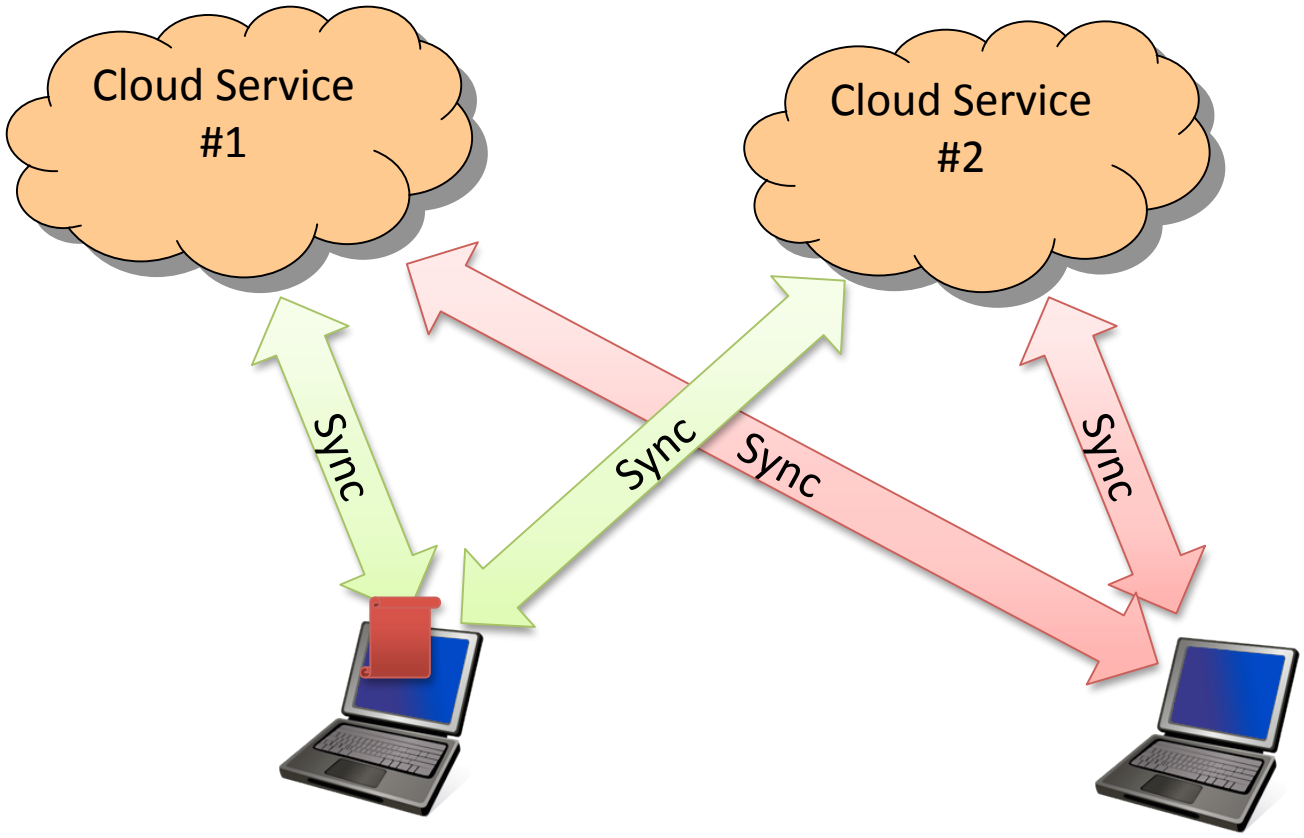
- Not detecting loses data
- Detecting "requires" histories?

No loops? Not sure!



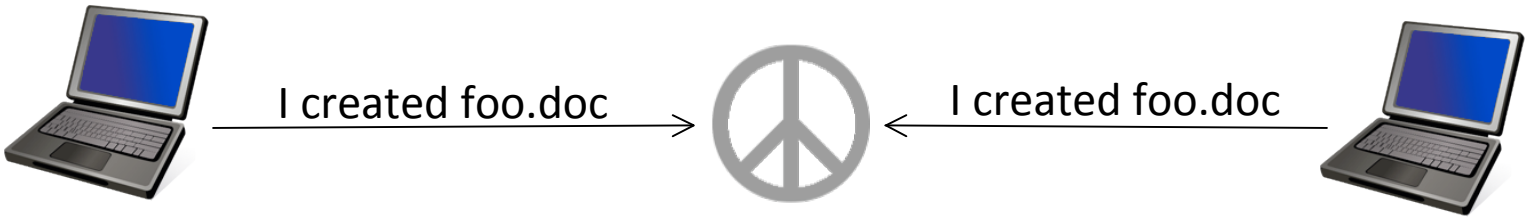
No one sync solution has loops, but taken together, they do!

No loops? Not sure!



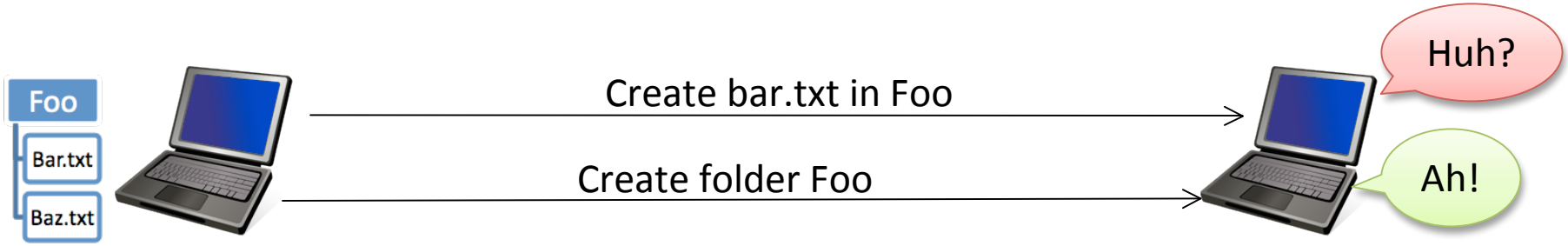
Syncing two devices to two services forms a loop!

Business Logic



- Synchronizing below app logic:
  - Ignores application semantics
  - Breaks apps in subtle and fragile ways
- Dealing with duplicate creates
  - Can lead to undesirable duplicates
  - Can lead to divergence

Hierarchy

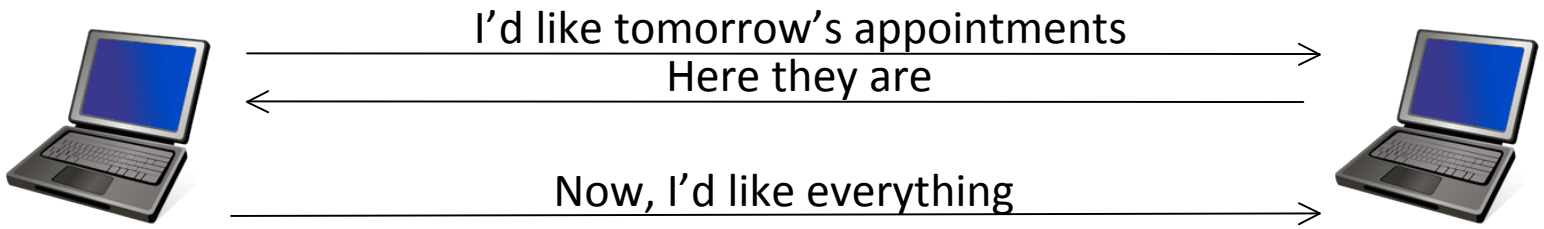


Problems:

- Sending changes out of order
  - Leads to inability to apply without retries
- Sending changes in hierarchy order
  - Leads to difficulties in handling interruptions

- Complex conflict scenarios
  - E.g. “Alice deletes Foo”, “Bob creates Bar in Foo”
  - Lead to data loss, bad user experience

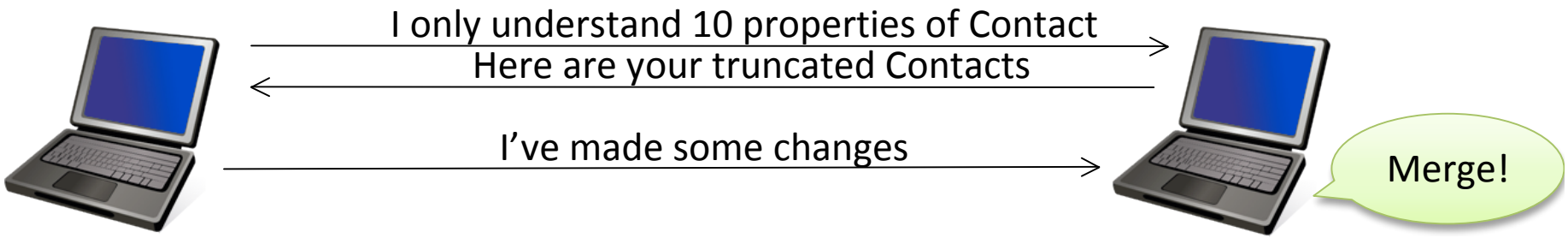
Item Filtering



Problems:

- Recording filters in sync watermark is difficult
- Desired filter can change over time
  - Need to send unchanged items
- Items move in and out of the filter
  - Sometimes without changes to the item itself, e.g.: "next two weeks of calendar"
- Replicas "forget" their own changes as they move out of filter
  - Then need to get them back from others

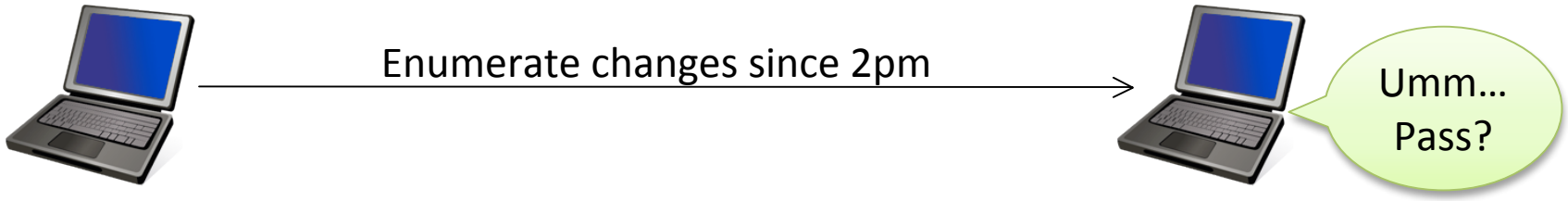
“Column” Filtering



Problems:

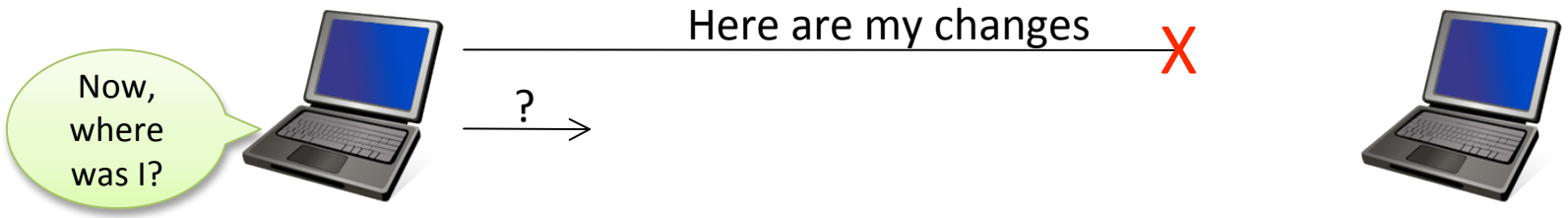
- Not resending things that were already sent
- Properly detecting conflicts
  - Where one of the endpoints hasn’t seen whole item
- Fidelity loss
  - E.g. pictures being down-scaled, or music being transcoded

Non-sync-enabled stores



- Stores often can’t store sync metadata
  - Can lead to fake conflicts (change reflection)
- Stores often can’t enumerate changes
  - Can lead to inability to sync
  - Can lead to really bad on-the-wire performance

Errors and interruptions

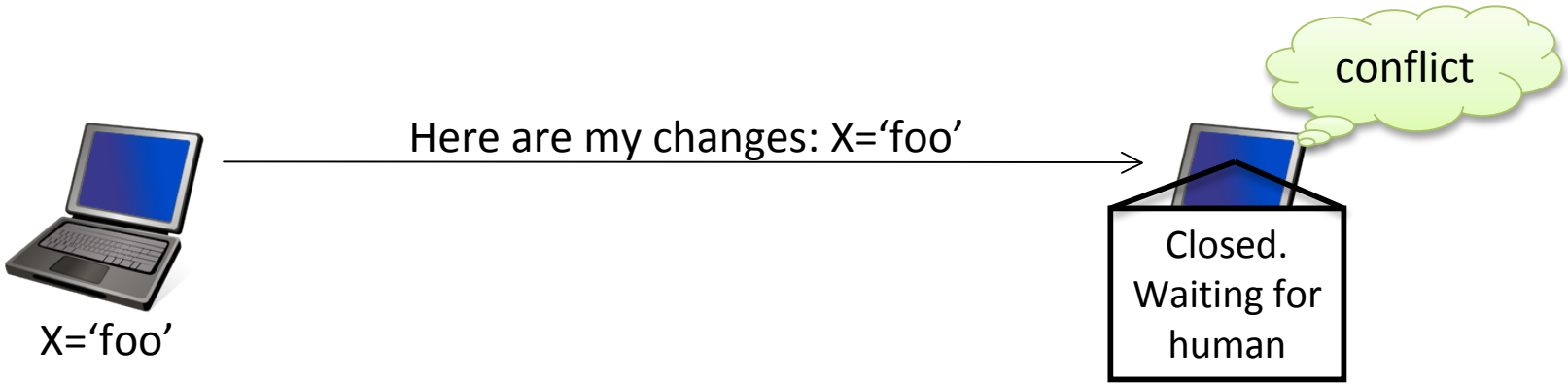


Problems:

- Watermark must describe errors
  - Otherwise: over-enumeration, etc
  - And do so without getting unreasonably large

- Sender never quite sure if receiver got it
  - Acknowledgements not reliable
  - Re-sending often unsafe

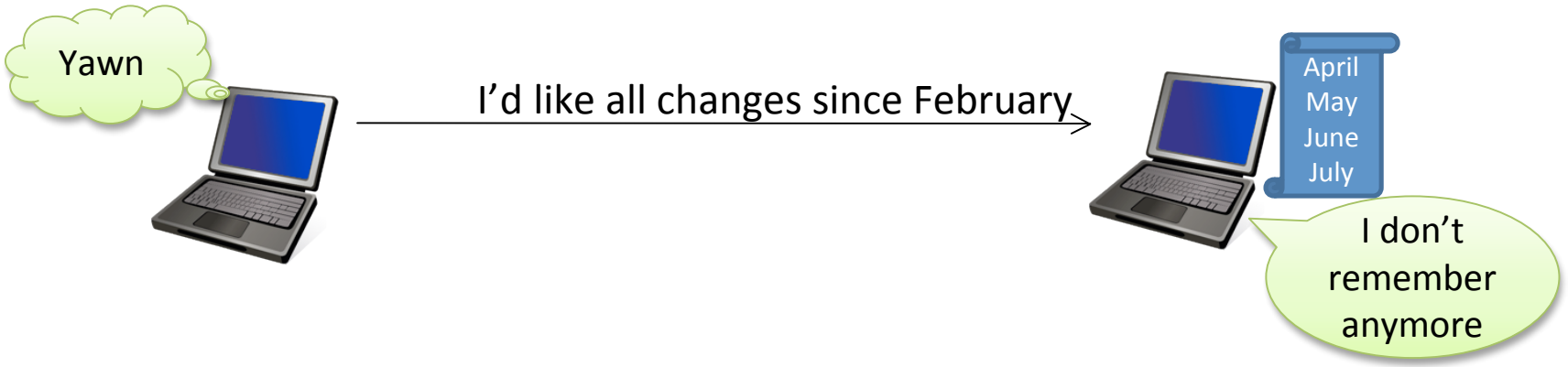
Deferred conflicts



Problems:

- Not all conflicts can be resolved automatically
  - Doing so can lose data
- Waiting for a human to resolve them
  - Locks devices and prevents progress
- Yet leaving them unresolved can lead to
  - Non-convergence or data loss
  - Sending the same data over and over
  - Having to resolve the same conflict many times

Loss of tombstones



- Tombstones need to be cleaned up
  - Otherwise unbounded meta-data growth
- Arrival of old replicas can cause
  - Undead items (non-convergence) or
  - Loss of data (re-init) or
  - Items coming back to life or
  - Lots of fake conflicts

MICROSOFT SYNC FRAMEWORK

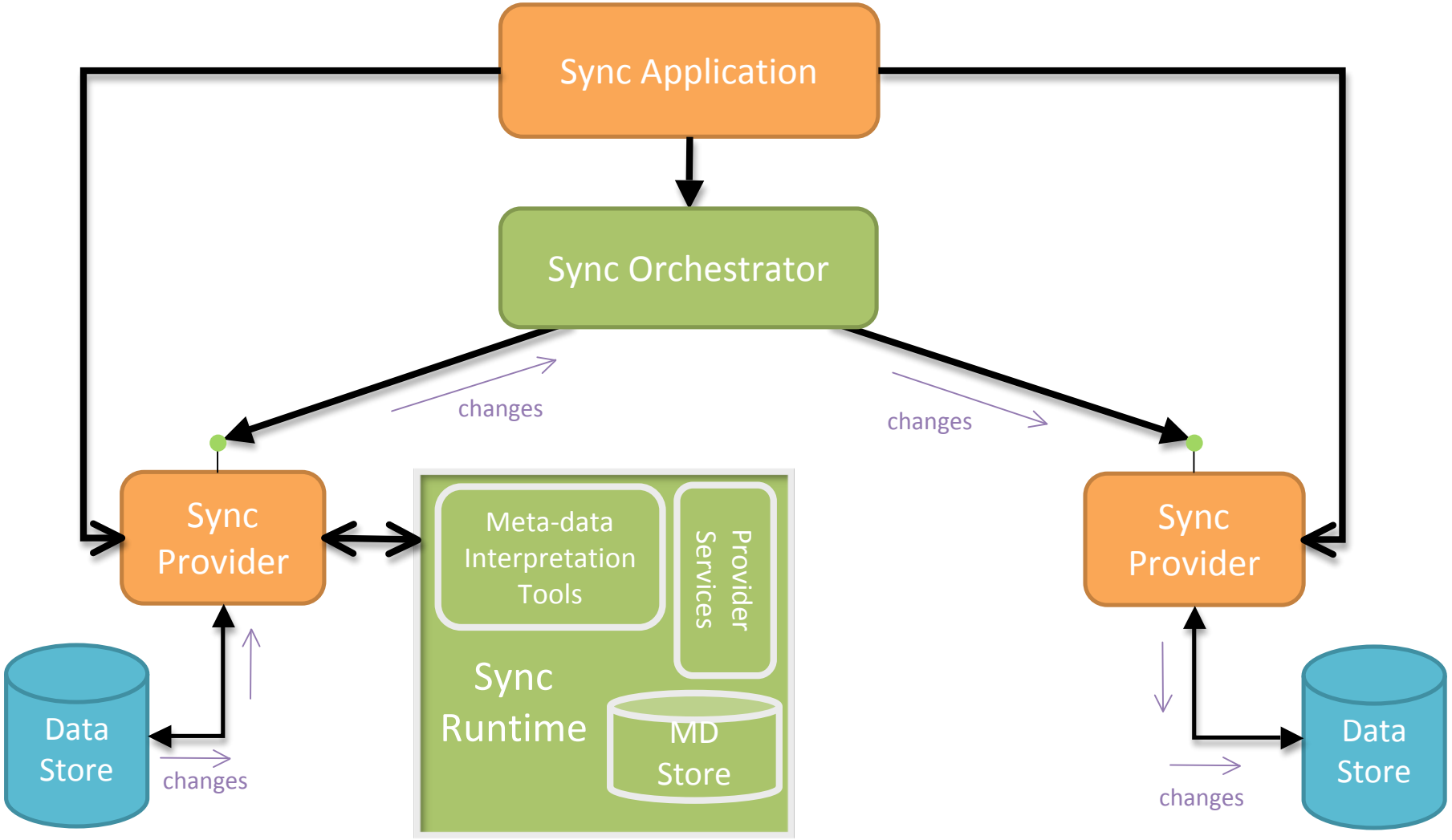
What does Microsoft Sync Framework do?

- Makes it easy for you to sync participating endpoints

- Comes with built-in endpoints:
  - V1: File System, Relational Databases
  - V2: SQL Data Services, Live Mesh, ADO.NET DS, and more
- Provides integrated tools and user experiences
  - Sync Services for ADO.NET
- Makes it easy for your endpoint to participate

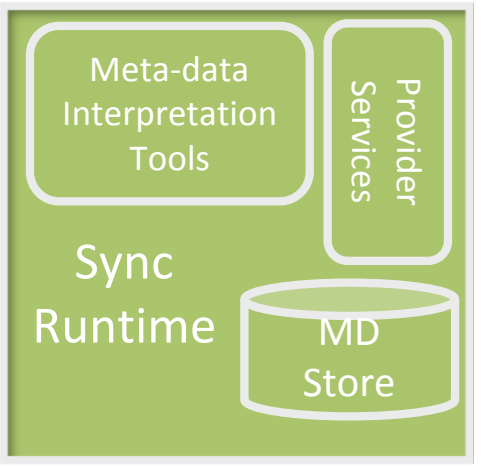
The Sync Session

Where Synchronization Happens



Sync Framework Runtime

- **Metadata** to:
  - Address (most of) aforementioned problems
  - Yet be concise (*knowledge, versions*)
- **Runtime** to:
  - Implement algorithms for aforementioned problems
  - Yet be store-, protocol- and type-independent
- **Metadata Store** to:
  - House metadata for those who can't do it themselves
- **Simple Provider Framework** to:
  - Make writing providers easy, offering perf←→complexity tradeoffs



Using Microsoft Sync Framework

What do customers do?

- Write **Sync Applications** to synchronize stores
  - Using built-in, other people's or your own providers
- Write **Providers** for your stores and apps
  - Using the Framework's Sync Runtime
  - Choose your balance of performance vs. complexity

Writing a Sync Application

```
SyncOrchestrator orch = new SyncOrchestrator();

orch.LocalProvider =
        new FileSyncProvider(guid1, “c:\\temp\\myfolder”);

 orch.RemoteProvider =
       new LiveMeshProvider(guid2,
                  new NetworkCreditial(username, password),
                  “MyMeshFolder”);

 orch.Synchronize();
```

v2 Simple Provider Framework

The easiest way to write simple providers

- Two flavors: full-enumeration and anchor-based
- Full-enumeration provider (think FAT) must:
  - Enumerate all objects in the scope
  - Find properties that tell objects apart
  - Find properties that tell if the object changed
  - Be able to Create, Delete, and Update objects
- Anchor-based providers (think NTFS) must:
  - Do all of the above
  - Enumerate all objects that changed since some point in the past

Provider Parts: schema

- Declare the properties identifying the objects
  - E.g. “Path”, “ID”, or a combination
- Declare the properties identifying the version
  - E.g. “Timestamp”, “Version”, or “Hash”

```
ItemMetadataSchema MetadataSchema { get
{
    var fieldPath = new CustomFieldDefinition(0, typeof(string), 256);
    var fieldLMT = new CustomFieldDefinition(1, typeof(UInt64));

    var identityRule =  new IdentityRule(new uint[] { 0 });

    var schema = new ItemMetadataSchema(
                    new CustomFieldDefinition[] { fieldPath, fieldLMT },
                    new IdentityRule[] { identityRule });
}}
```

Provider Parts: enumeration

Full Enumeration:

Iterate over the entire scope, return all objects in the form of ItemFieldDictionary

```
IEnumerable<ItemFieldDictionary> EnumerateItems(FullEnumerationContext)
{
    var items = new List<ItemFieldDictionary>();

    foreach(Contact c in this.contacts)
         items.Add(c.ToDictionary());
    return items;
}
```

Anchor-model

Enumerate all changes since your last anchor (anchor), and returning an updated anchor

```
IEnumerable<LocalItemChange> EnumerateChanges(
    byte[] anchor,
    AnchorEnumerationContext context,
    out byte[] updatedAnchor)
```

Provider Parts: operations

```
void InsertItem(object itemData,
            RecoverableErrorReportingContext recoverableErrorReportingContext,
            out ItemFieldDictionary keyAndUpdatedVersion)
```

Create specified object, **return** identifying attributes (e.g. Path, LMT) as an `ItemFieldDictionary`

```
void UpdateItem(object itemData,
            IEnumerable<SyncId> changeUnitsToUpdate,
            ItemFieldDictionary keyAndExpectedVersion,
            RecoverableErrorReportingContext recoverableErrorReportingContext,
            out ItemFieldDictionary keyAndUpdatedVersion)
```

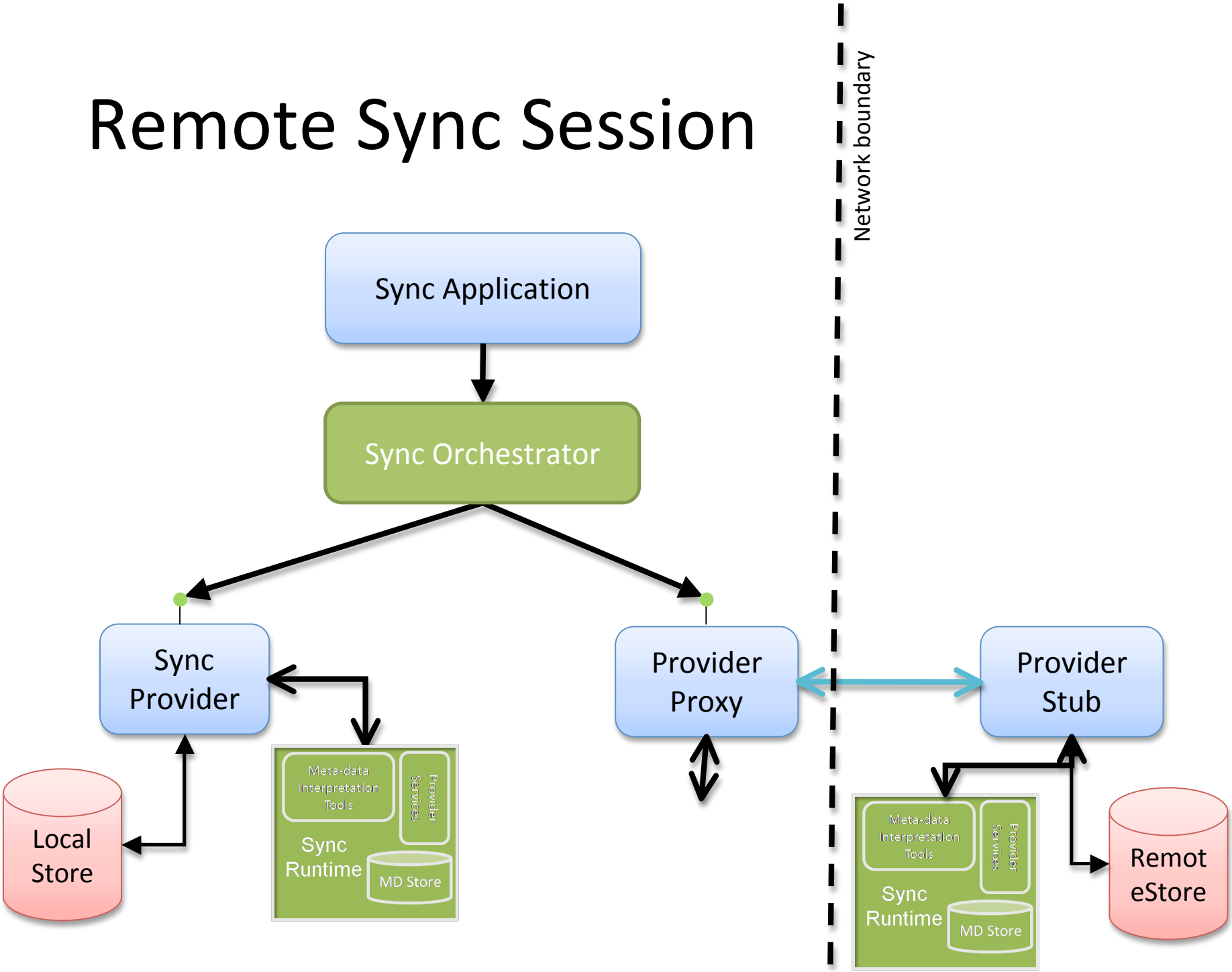
Find object by its key; **ensure** it hasn't changed since expected version; **update** it; **return** new identifying attributes (as `ItemFieldDictionary`)

```
void DeleteItem(ItemFieldDictionary keyAndExpectedVersion,
            RecoverableErrorReportingContext recoverableErrorReportingContext)
```

Find object by its key; **ensure** it hasn't changed since expected version; then **delete** it

PARTIAL PARTICIPANTS

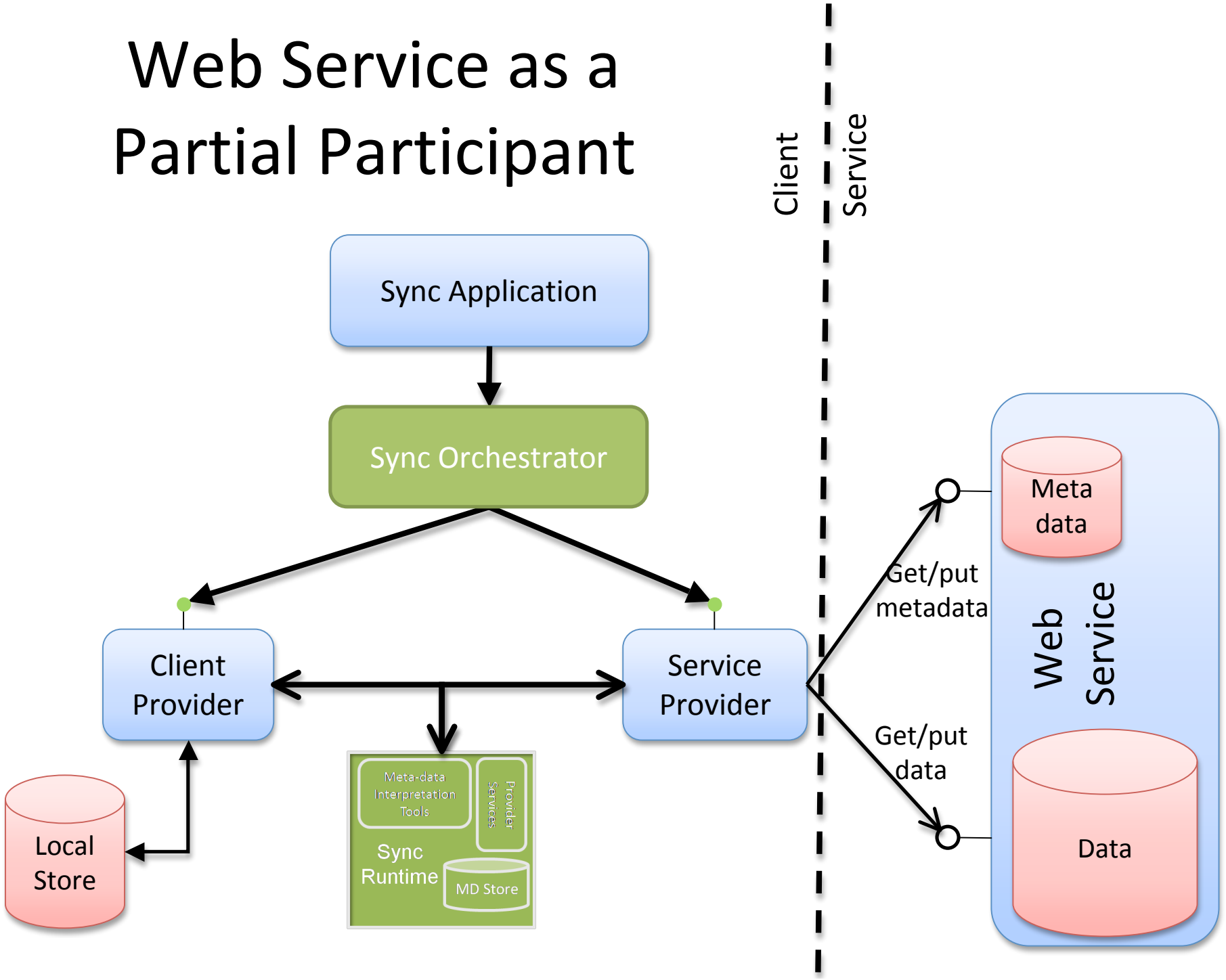
Remote Sync Session



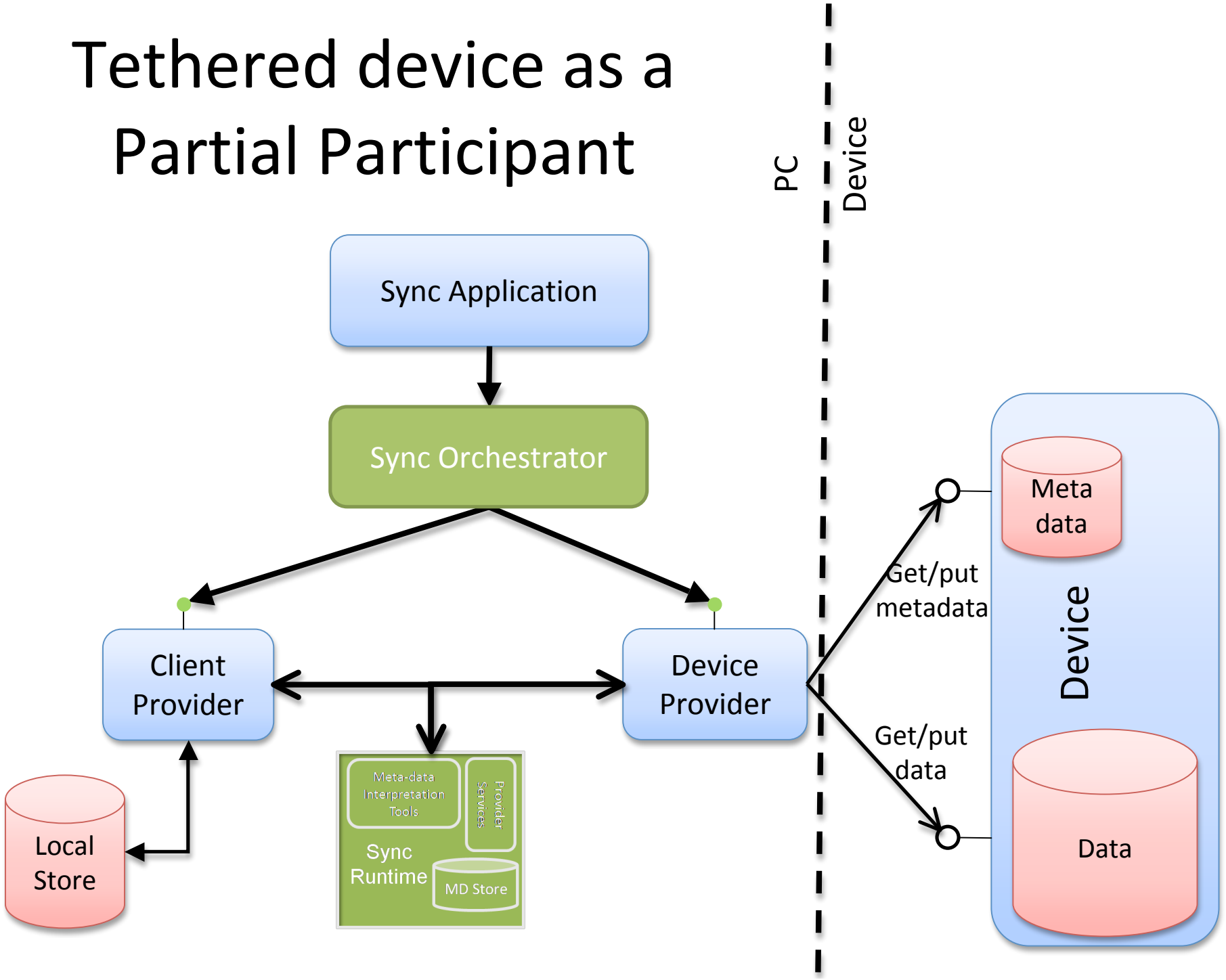
Partial Participants

- Sync Endpoints that:
  - Store metadata (in an agreed-upon way)
  - Do minimal processing on update
  - Optionally, simple change enumeration
- Rest is done by the other side
  - Sync Framework supports the pattern
  - Provider needs to write the code

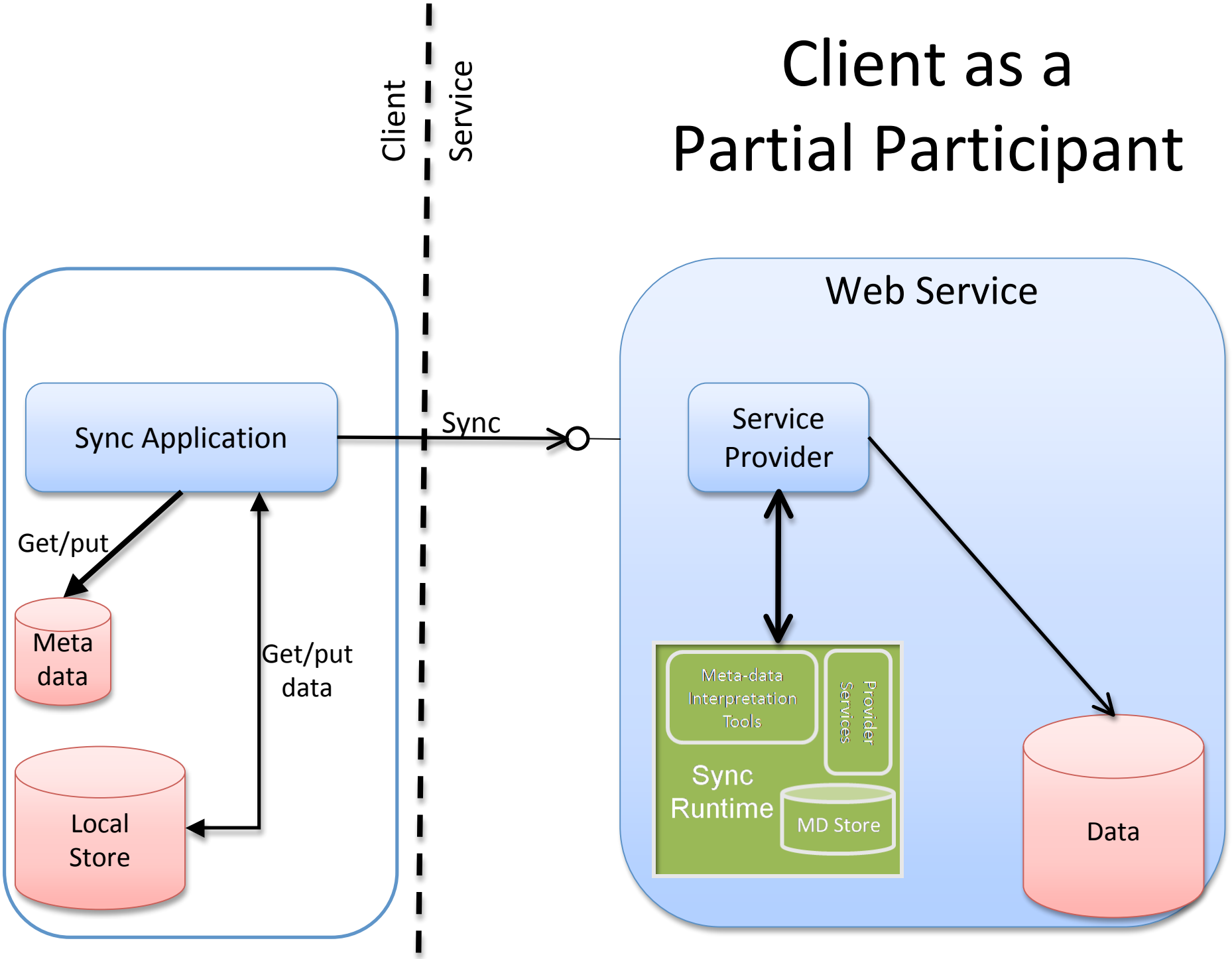
Web Service as a Partial Participant



Tethered device as a Partial Participant

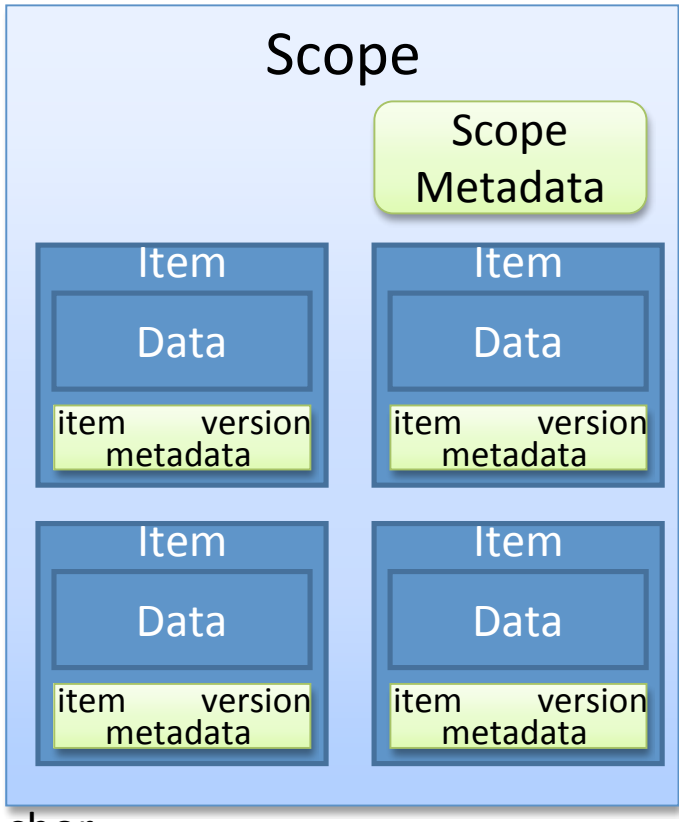


Client as a Partial Participant



Making a Web Service into a Partial Participant

- Pick a scope of synchronization
  - E.g.: Photo Album
- Per scope: **one piece of metadata**
  - No logic, pure get and set
  - Can be large; usually small
- Per-item: two pieces of metadata
  - **Item metadata**: Get/Set, don't touch
  - **Version metadata**: Get/Set, set to null on update
  - Set **atomically** with item operations
- Change enumeration
  - If supported, return changes since **anchor**, plus anchor
  - If not keeping tombstones
    - Return **count** of items in the scope



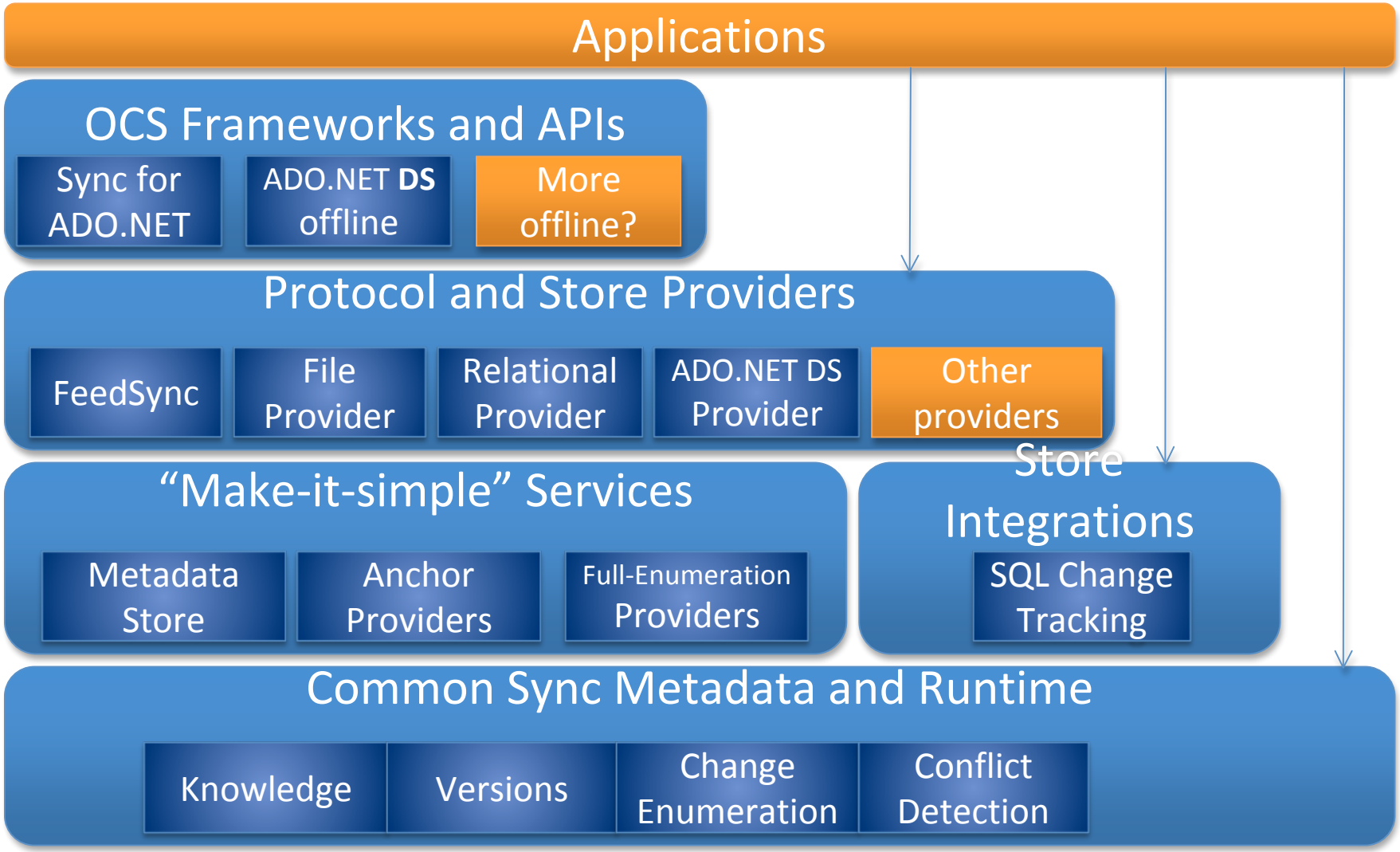
Provider Design sketch

- In Scope Metadata, store:
  - Knowledge, Forgotten Knowledge
- In Item Metadata, store:
  - Creation Version, Update Version
- In Version Metadata store:
  - Update Version
  - To compute real Service Update Version
    - If Version Metadata matches Item Metadata: take Item Metadata's
    - Otherwise, allocate a new one (and store it)
- *This is an over-simplification*

Other v2 features

- Filtering (column, item, forgetting)
- Constraint conflicts (moving, merging)
- Metadata store improvements
- Improved push-model support

Overall Layering



Sync Framework Resources

- Sync Developer Center
  http://msdn.microsoft.com/sync
  - SDK (including documentation)
  - Samples (including several end-to-end)
- Sync Blog
  http://blogs.msdn.com/sync
  - Announcements, Tips and Tricks
- Me
  - levn@microsoft.com

QUESTIONS?

Backup

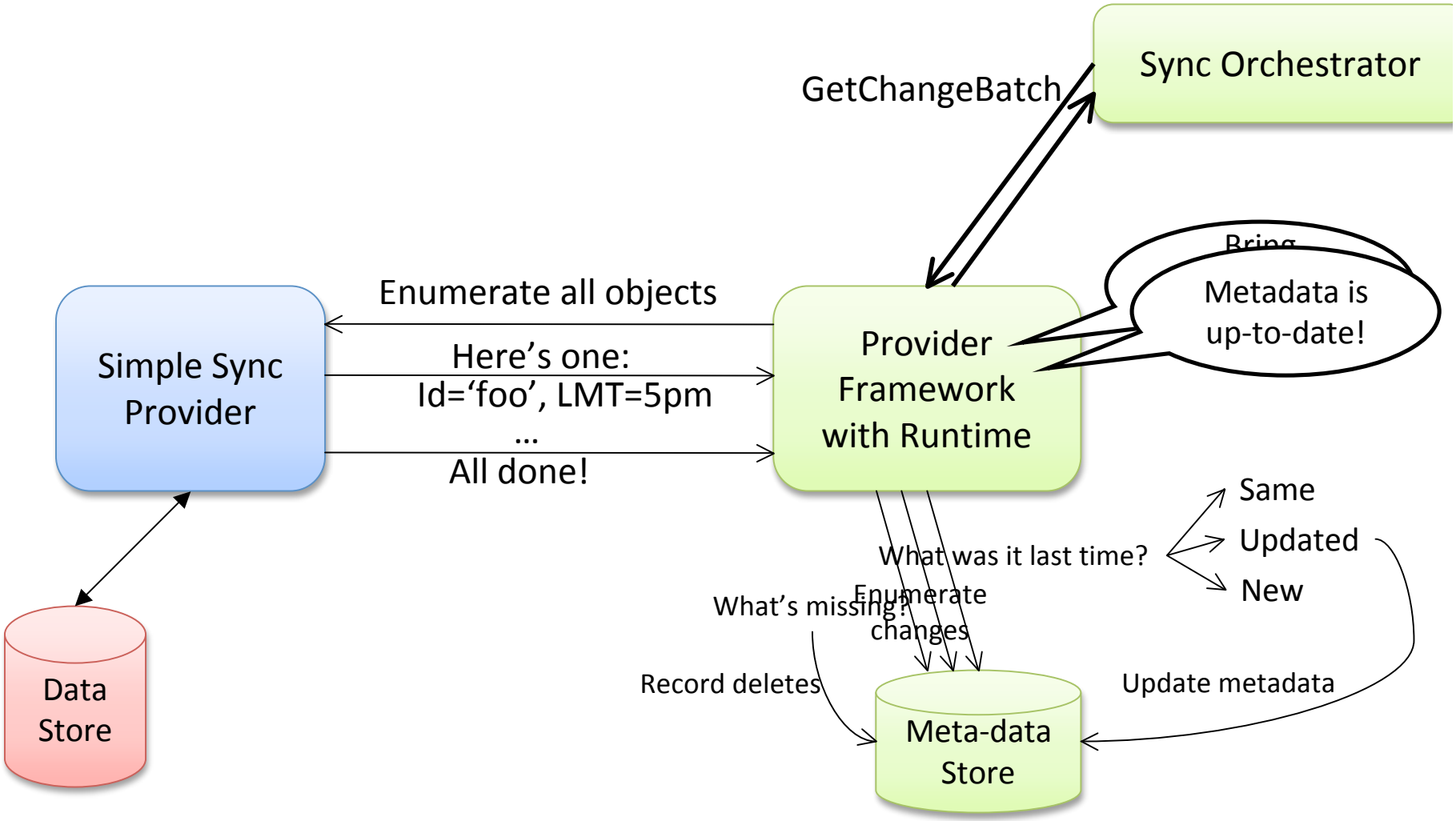
Writing Providers

Responsibilties

- Store responsibilities:
  - Track changes
  - Store metadata
- Provider responsibility: expose store's capabilities
  - Detect local changes
  - Enumerate changes when requested
  - Apply changes and record metadata
- Some stores have no such abilities, e.g. FAT
  - No place to store metadata
  - No ability to track changes
- What is a provider to do?

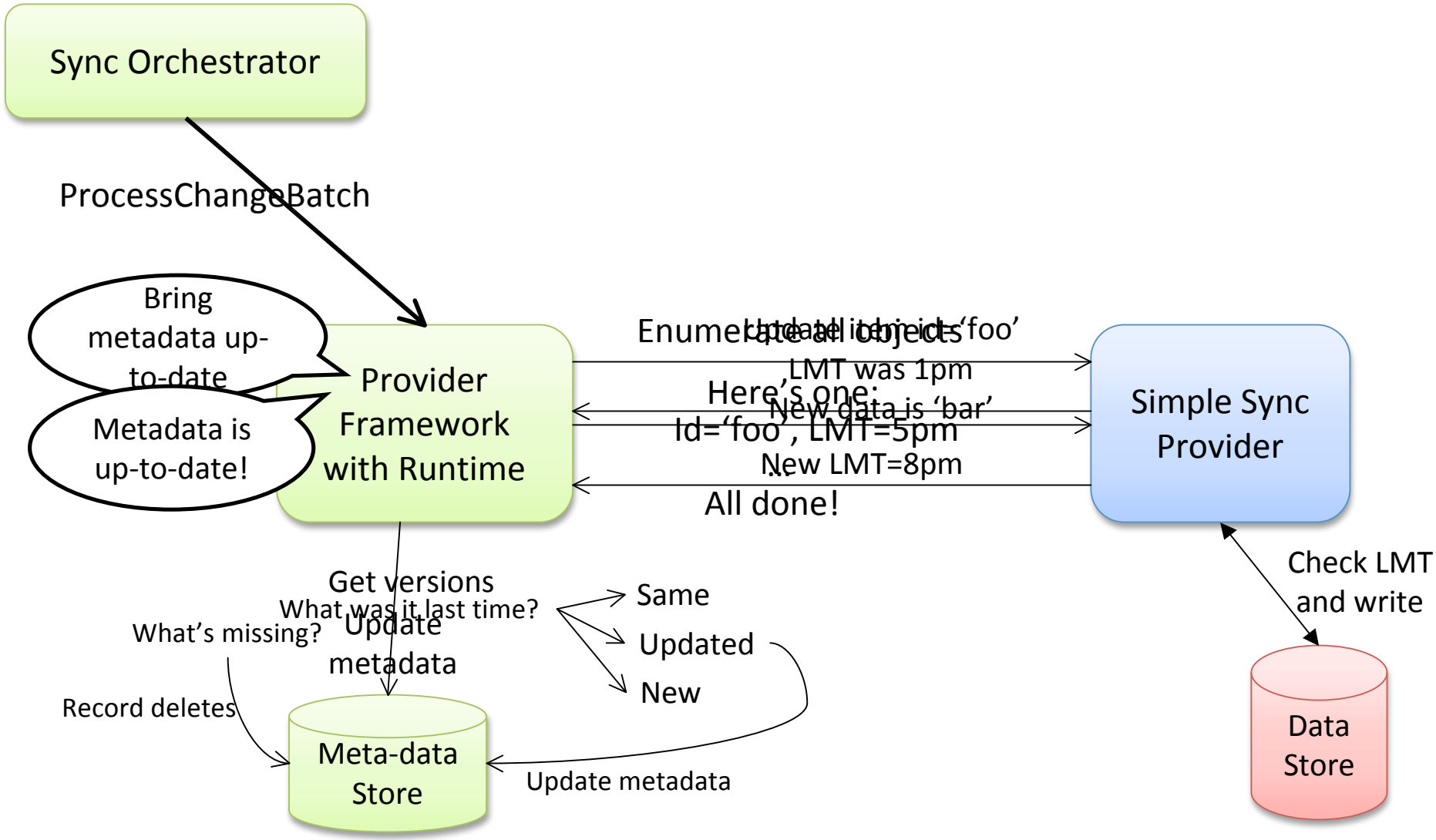
How Provider Framework works

Under the covers: Enumeration



How Provider Framework works

Under the covers: Applying changes

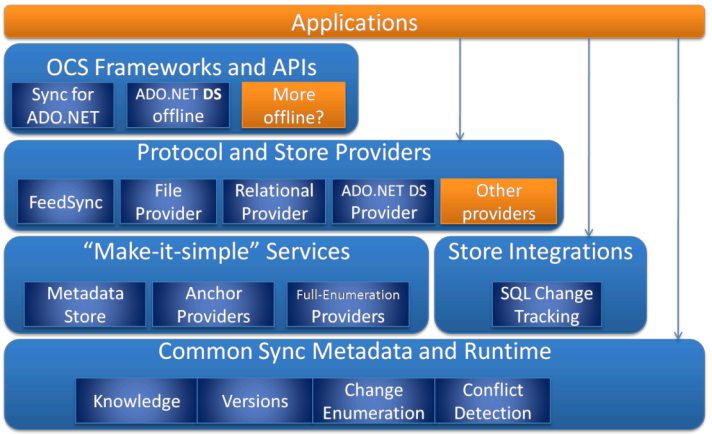


Targeted Infrastructure

- **Relational** Provider
  - Enables applications built on relational databases to easily sync with each other
  - Integration Through ADO.Net Commands
  - Builds on the OCS Functionality Delivered in Orcas
- **File** System Provider
  - Support Simple File Systems Like FAT32/TFAT
  - Handle Complex Issues Associated with Hierarchy
  - Basis for **SyncToy** 2.0
- **FeedSync** Support

Summary: varying entry points

- "I need to cache my (service) data offline"
  - Cache it in SQL-CE using ADO.NET Sync Services
- "No, I need to sync particular stores"
  - Use Sync Providers for those stores
  - Use Orchestrator to orchestrate



- "But how do I communicate my changes remotely?"
  - Use FeedSync support to generate and consume feeds
  - Alternatively, use an optimized protocol (and still be compatible)
- "But there is no provider for this store"
  - Write one easily using Simple Provider models and metadata store
- "I need better performance and integration"
  - Use Knowledge Services to store metadata yourself