CRDTs, Yjs and pycrdt

David Brochart - QuantStack

CRDT: Concurrently Replicated Data Type

Data type (JSON-like):

- Text
- Array
- Map

Replicated:

- Data doesn’t live in a central place: distributed vs centralized

Concurrently:

- Data can be changed “at the same time”, on different machines
- Changes are made to all data replicas, ensuring the same state

CRDTs: data structures with superpowers

Data changes can:

- be observed (callback with change event)
- be serialized to binary
- travel over the wire (WebSocket, WebRTC, etc.)
- be applied on the other end of the wire

To end users, CRDTs are very simple:

- Create a shared object
- Connect it through a “provider”
- React to changes
- Lock-free shared data

Do we need CRDTs?

A simple example without CRDTs:

- Two users work on a shared document, starting with an blank page
- User A types “a” and user B types “b” at the same time:
  - User A’s change is: insert “a” at index 0
  - User B’s change is: insert “b” at index 0
- Before changes propagate:
  - User A has “a”
  - User B has “b”
- After changes propagate:
  - User A has “ba”
  - User B has “ab”
- Documents have diverged

Do we need CRDTs?

We could have users A and B access the same document on a server:

- Users post desired changes, applied changes are echoed back
- User A posts change: insert “a” at index 0
- User B posts change: insert “b” at index 0
- Users A and B receive change: insert “a” at index 0
- Users A and B receive change: insert “b” at index 0
- Users A and B have: “ba”

Okay, but latency between desired changes and applied changes (UX).

Also: what if connection breaks? What if distributed architecture?

CRDTs: a paradigm shift

Local-first applications:

- The application owns the data: no latency
- Data is synced in the background
- The application reacts to data changes
- A server can still be used, but it’s not mandatory (can work offline)

The application is not anymore a master sending changes to a server, it’s a slave receiving changes, and it must update accordingly.

CRDT implementations

“Y” family:

- Yjs (JavaScript)
- Yrs (Rust)
- Ywasm (WebAssembly)
- pycrdt (Python bindings to Yrs)

Automerge:

- JavaScript
- Rust

Loro

CRDTs in JupyterLab

In the frontend:

- Yjs: JavaScript implementation of CRDTs
- jupyter-ydoc: shared documents types for Jupyter (text, notebook…)
- jupyter-collaboration: WebSocket client

In the backend:

- pycrdt: Python bindings for Yrs, the Rust port of Yjs
- jupyter-ydoc: shared documents for Jupyter (text, notebook…)
- jupyter-collaboration: WebSocket server
- Notion of “rooms” where users share a document
- Synchronization with file (auto-save, out-of-band changes)

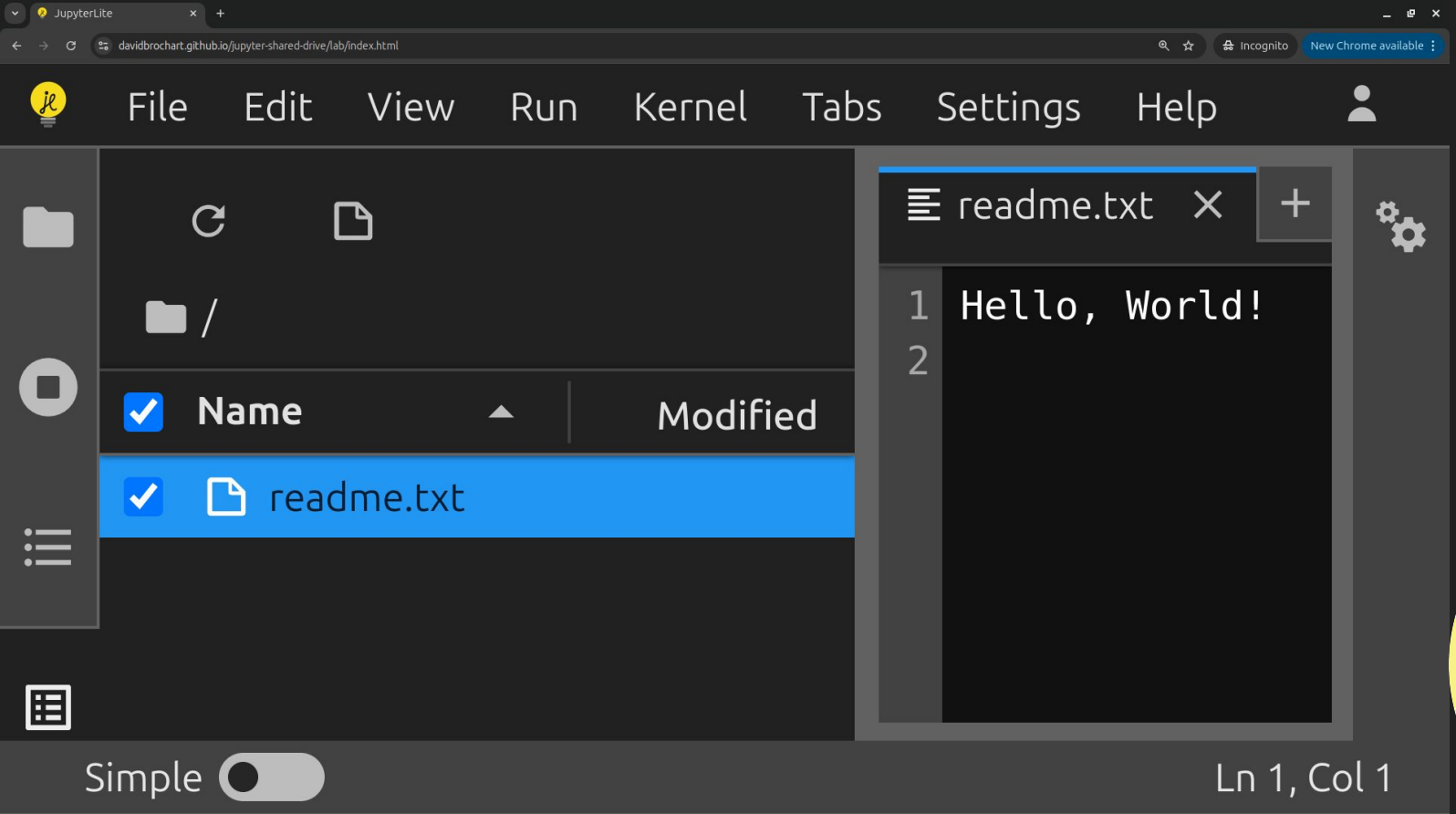
CRDTs in JupyterLite (new!)

No backend!

In the frontend:

- Yjs: JavaScript implementation of CRDTs
- jupyter-ydoc: shared documents for Jupyter (text, notebook…)
- jupyter-shared-drive: WebRTC client
  - The drive doesn’t replace the default drive (think scratch disk)
  - Files can be imported from the default drive
  - Distributed, peer-to-peer architecture
  - The filesystem is itself a shared document
  - Each client has a “copy” of the files
  - Ephemeral documents, exist as long as there is at least one client

https://davidbrochart.github.io/jupyter-shared-drive



Show me some code!

https://jupyter-server.github.io/pycrdt

```python
from pycrdt import Text, Array, Map

text0 = Text("Hello")
array0 = Array([0, "foo"])
map0 = Map({"key0": "value0"})
```

```python
from pycrdt import Doc

doc = Doc()
doc["text0"] = text0
doc["array0"] = array0
doc["map0"] = map0
```

Changing and composing data

```
text0 += ", World!"
array0.append("bar")
map0["key1"] = "value1"
```

```
map1 = Map({"baz": 1})
array1 = Array([5, 6, 7])

array0.append(map1)
map0["key2"] = array1
```

Document updates

```
update = doc.get_update()

# the (binary) update could travel on the wire to a remote machine:

remote_doc = Doc()
remote_doc.apply_update(update)

remote_doc["text0"] = text0 = Text()
remote_doc["array0"] = array0 = Array()
remote_doc["map0"] = map0 = Map()
```

Transactions

```
with doc.transaction():
    text0 += ", World!"
    array0.append("bar")
    map0["key1"] = "value1"
```

```
with doc.transaction() as t0:
    text0 += ", World!"
    with doc.transaction() as t1:
        array0.append("bar")
        with doc.transaction() as t2:
            map0["key1"] = "value1"
```

Blocking transactions: using multithreading

```python
from threading import Thread
from pycrdt import Doc

doc = Doc(allow_multithreading=True)

def create_new_transaction():
    with doc.new_transaction(timeout=3):
        ...

t0 = Thread(target=create_new_transaction)
t1 = Thread(target=create_new_transaction)
t0.start()
t1.start()
t0.join()
t1.join()
```

Blocking transactions: using asynchronous programming

```python
from anyio import create_task_group, run
from pycrdt import Doc

doc = Doc()

async def create_new_transaction():
    async with doc.new_transaction(timeout=3):
        ...

async def main():
    async with create_task_group() as tg:
        tg.start_soon(create_new_transaction)
        tg.start_soon(create_new_transaction)

run(main)
```

Shared data events

```
from pycrdt import TextEvent

def handle_changes(event: TextEvent):
    # process the event
    ...

text0_subscription_id = text0.observe(handle_changes)
```

```
text0.unobserve(text0_subscription_id)
```

Observing nested changes: deep observability

```
from pycrdt import ArrayEvent

def handle_deep_changes(events: list[ArrayEvent]):
    # process the events
    ...

array0_subscription_id = array0.observe_deep(handle_deep_changes)
```

Document events

```python
from pycrdt import TransactionEvent

def handle_doc_changes(event: TransactionEvent):
    update: bytes = event.update
    # send binary update on the wire

doc.observe(handle_doc_changes)
```

```python
# receive binary update from e.g. a WebSocket
update: bytes

remote_doc.apply_update(update)
```

Undo manager

```python
from pycrdt import Doc, Text, UndoManager

doc = Doc()

text = doc.get("text", type=Text)
text += "Hello"

undo_manager = UndoManager(doc=doc)
undo_manager.expand_scope(text)

text += ", World!"
print(str(text))
# prints: "Hello, World!"

undo_manager.undo()
print(str(text))
# prints: "Hello"

undo_manager.redo()
print(str(text))
# prints: "Hello, World!"
```