Server/Client model for ROOT7 graphics

Sergey Linev,
GSI, Darmstadt,
26.04.2017

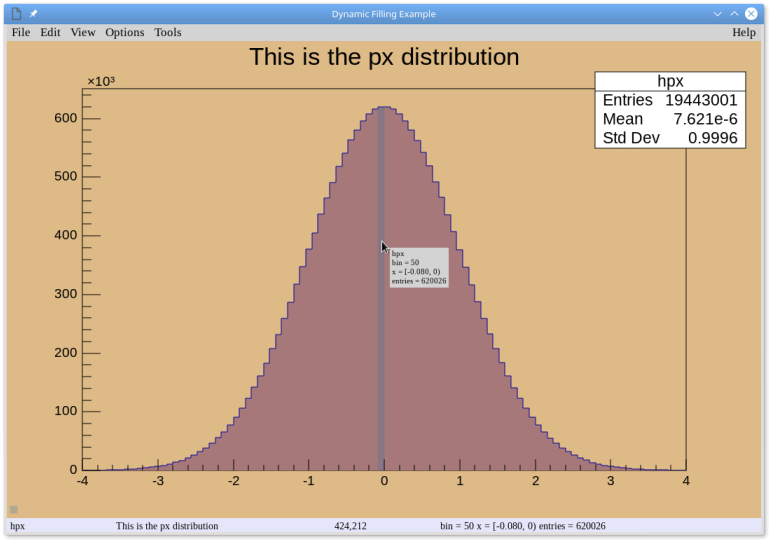
Server/Client model

- Server:
  - ROOT C++ application producing data
- Client:
  - JavaScript producing graphical output
- Communication:
  - websocket-based protocol

Server/Client model

```
auto h1 = new Histogram;
auto c1 = new Canvas;
c1->Add (h1, “hist”);
c1->Draw();
while (flag) {
    h1->Fill(random());
    c1->Modified();
    c1->Update();
}
```

websocket



Server
plain ROOT C++ code

Client
any web browser

Client side

- JavaScript ROOT as code base
  - SVG graphics for 2D
  - WebGL for 3D
  - mixture on the same canvas is possible
- Display, update and interact with data
- Image production
- Integration with UI frameworks

Communication

- websocket
  - implemented in **civetweb** (THttpServer)
  - supported by all web browsers
  - long polling (async http) as fallback solution

- Custom text-based protocol
  - server -> client: mainly object data as JSON
  - client -> server: commands or notification events
  - not directly accessible for the users

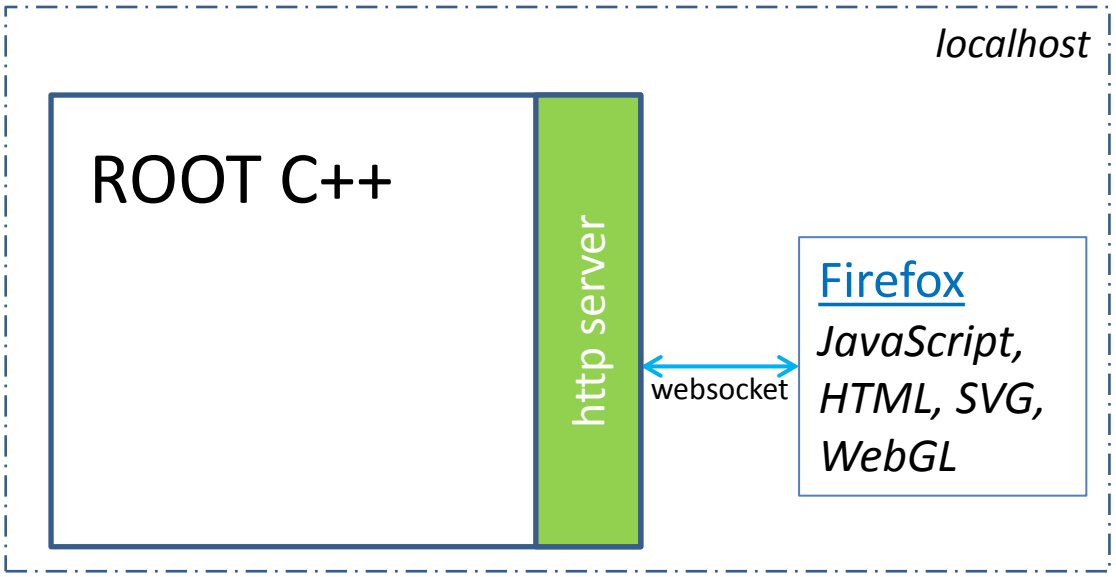
- Required unique objects IDs:
  - to correctly update objects drawing
  - to get list of context menu items
  - to execute objects method

Server side

- Keep drawing logic of ROOT C++ code

- Local and remote displays
- Multiple clients for the same canvas
- Batch mode
- Multithreading

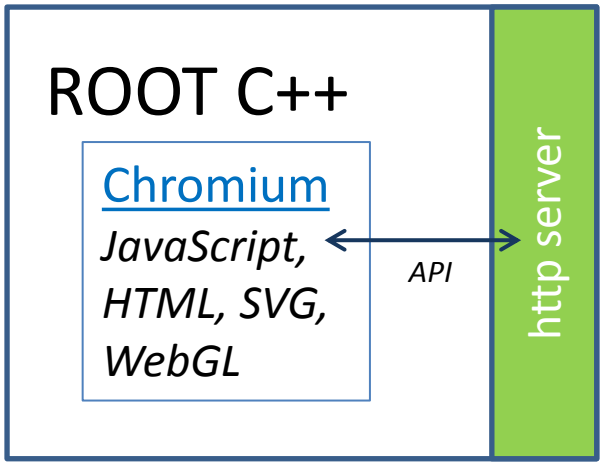
Local clients



http://localhost:8080/Canvases/c1/draw.htm?websocket

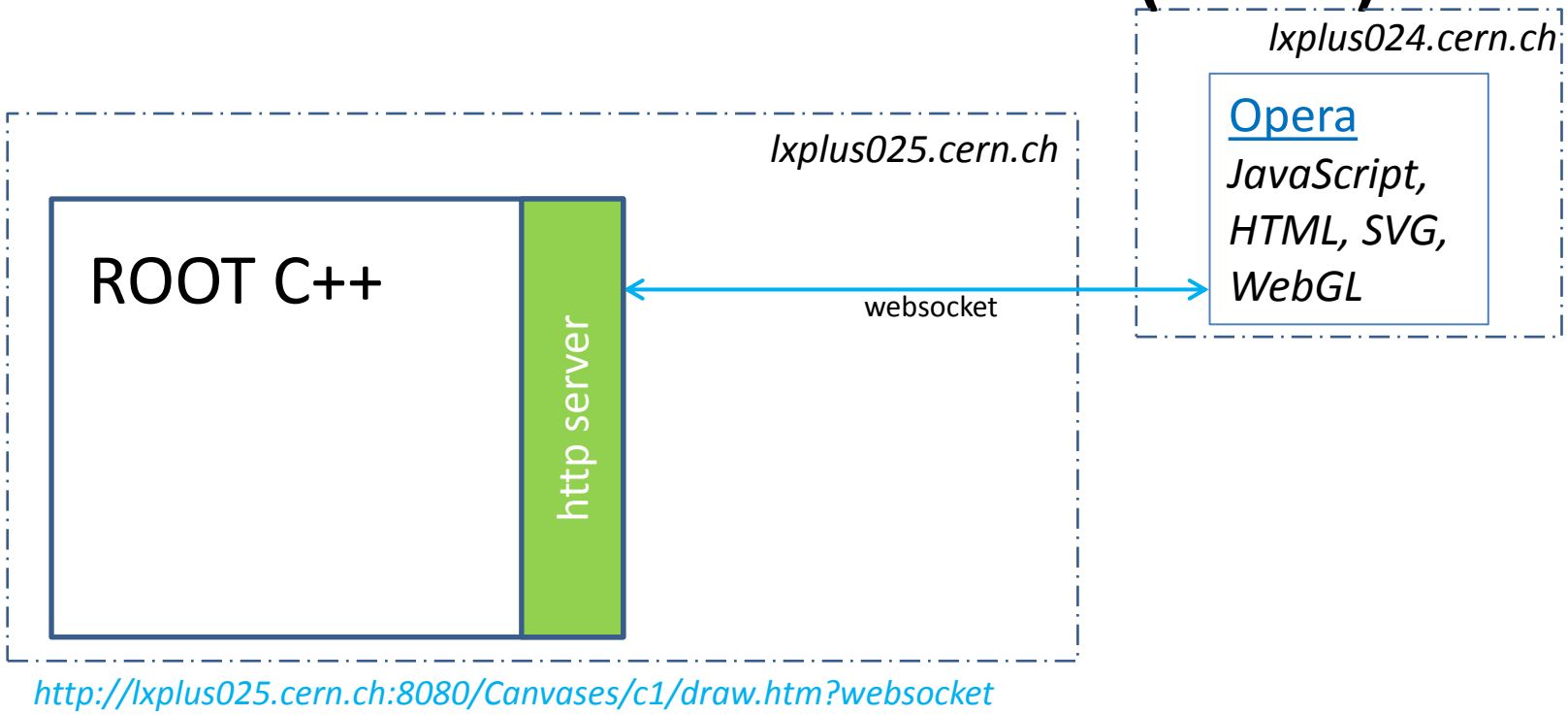
- Run any browser on the some host
- Use THttpServer bound to loopback address
- Communication via websockets

Local clients



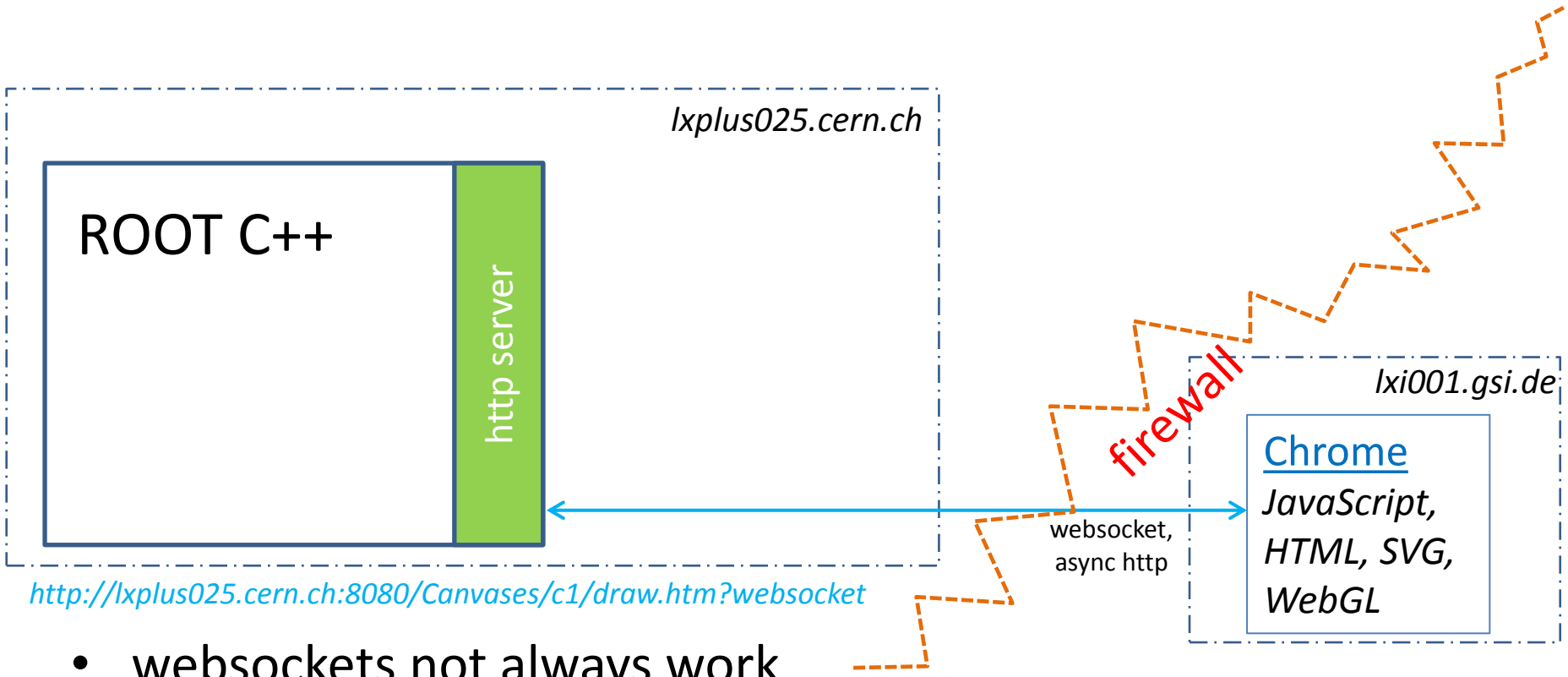
- Use Chromium Embedded Framework CEF
- Create necessary window(s) directly in C++
- Communication via CEF API - no any sockets

Remote clients (LAN)



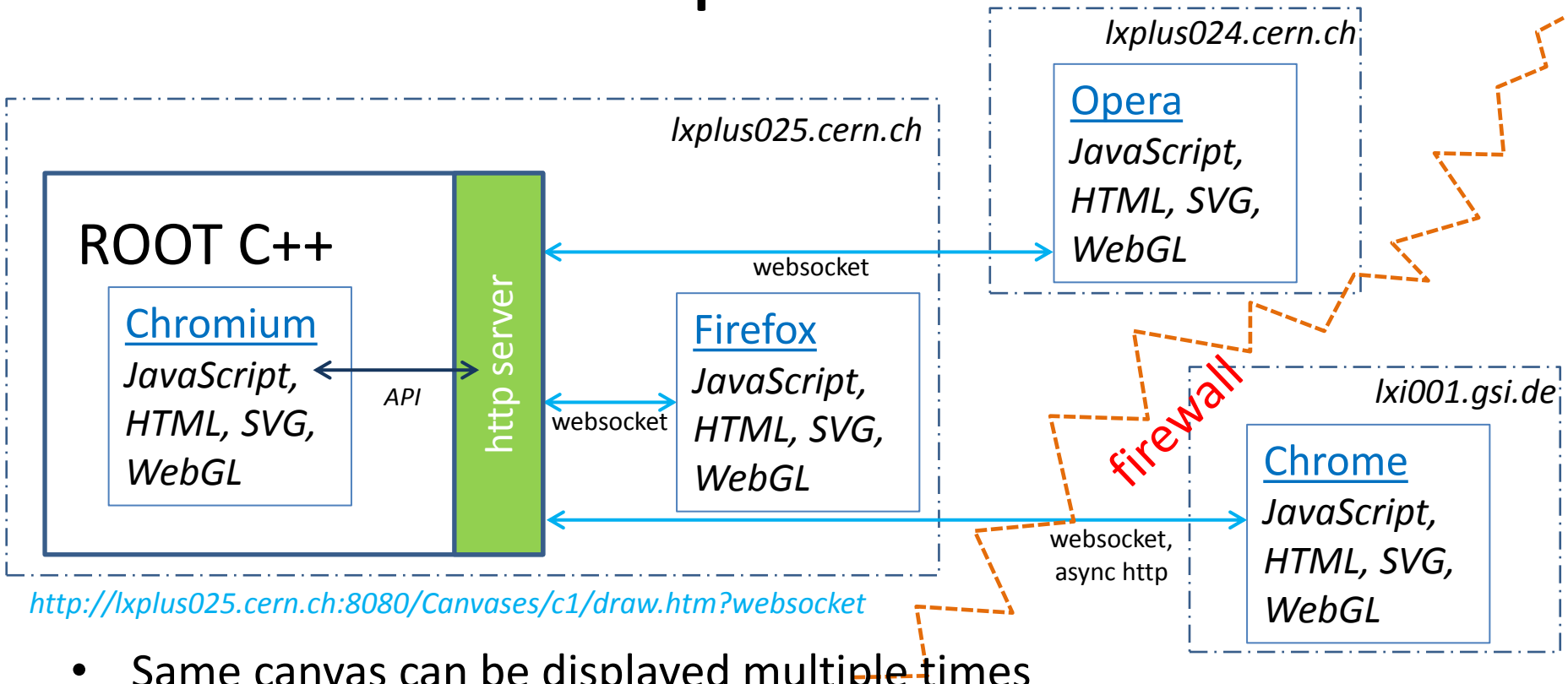
- No any difference with local clients
- Bound THttpServer with normal IP address
- Communication via websockets

Remote clients (WAN)



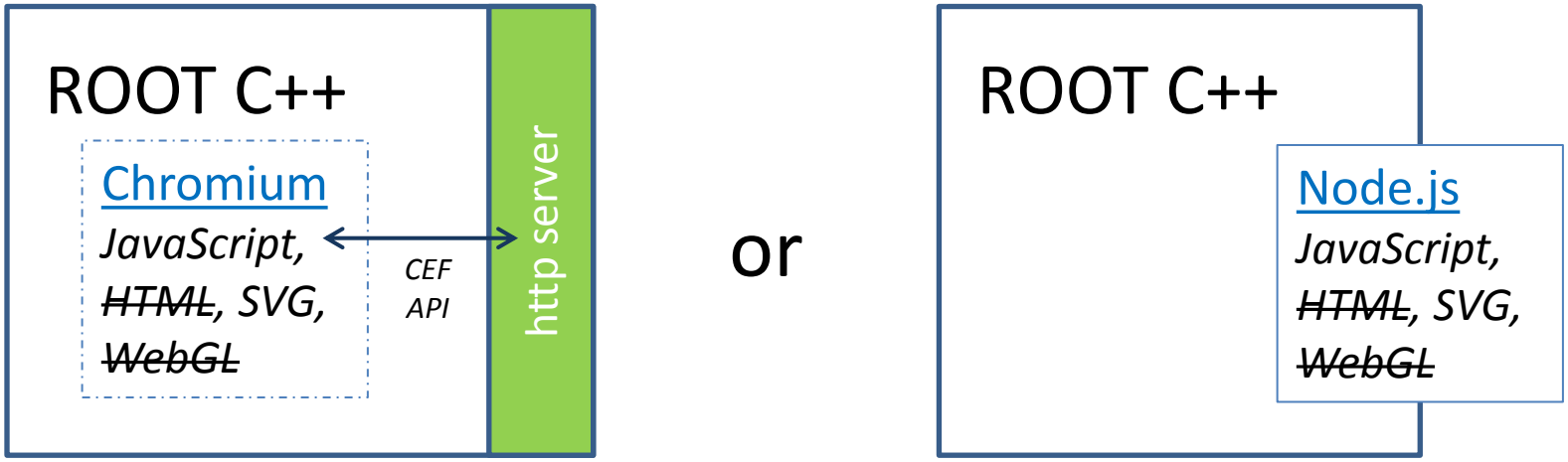
- websockets not always work
  - firewall/proxy limitation
  - not supported in fastcgi
  - automatic fall back to long polling (async http)

Multiple clients



- Same canvas can be displayed multiple times
- Separate websocket for each client
  - different access rights for different clients
- Not all changes in clients propagated to the server
  - like zooming can be done independently

Batch mode



- For image production
  - SVG, PS, PNG, GIF, ...
- No any window created
- No any socket opened
- Instead of WebGL use three.js SVG renderer
  - not very performant, need to be improved

User API

- Basics remain the same:
  - there will be *Canvas* class
  - canvas will have list of primitives with draw options
  - canvas can have sub-pads
  - ROOT histograms and graphs classes will work as before

- That will be changed:
  - there is no gPad and many other global pointers
  - graphical attributes will be separated from data

- Main change – Paint() method
  - not painting, but preparing data for JavaScript
  - one could provide object itself or any alternative data
  - actual paint code and interactivity in .js scripts
  - as fallback, one could use PaintLine(), PaintBox(), ... methods
    - not very efficient and without much interactivity

Paint examples

```
void Graph::Paint(PadPainter* painter)
{
   // use graph object as is
   // painter already exists in JavaScript code
   painter->Draw(this);
}

void Histogram::Paint(PadPainter* painter)
{
   FillBins(); // call any method before paint
   painter->Draw(this, ”JSRootHistPainter.js”);  // provide script name
}
```

Paint examples

```
void PaveStats::Paint(PadPainter* painter)
{
   HistStats* stats = fHist->CreateStatistic(); // create special object with stats
   painter->DrawModel(stats);  // provide model, set ownership
}
```

```
void Box::Paint(PadPainter* painter)
{
   // JavaScript will covert it into SVG
   painter->PaintBox(fX1, fY1, fX2, fY2);
}
```

Paint examples

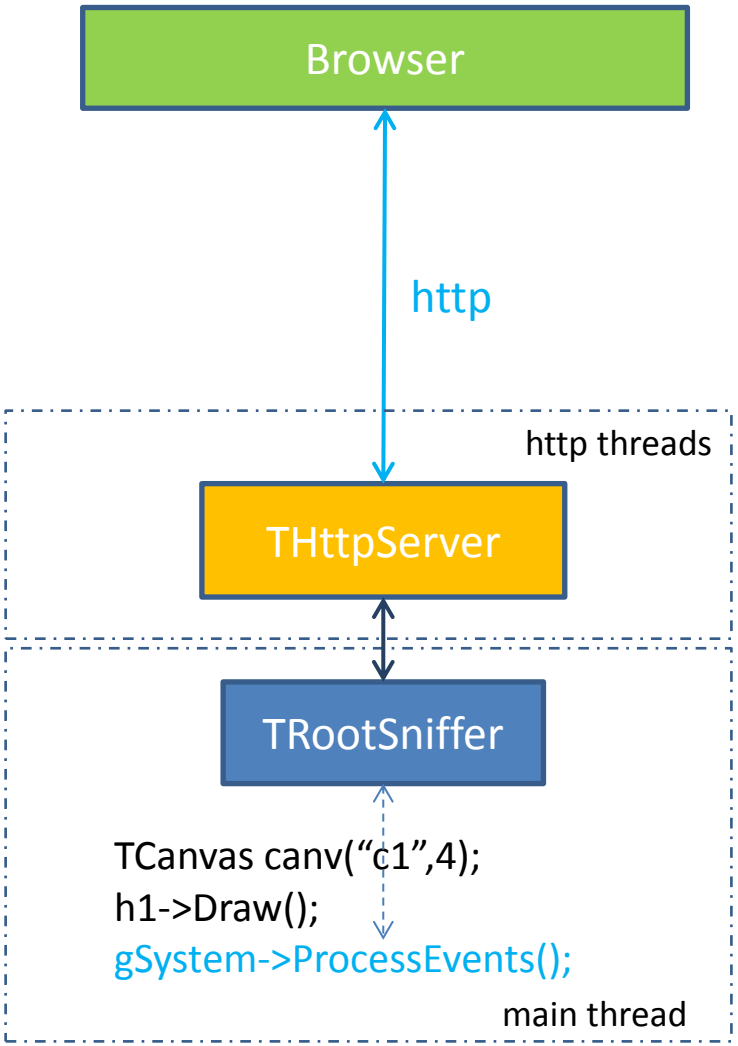
```
void UserClass::Paint(PadPainter* painter)
{
   std::string jsfunc =
       “function(obj) {“
       “  var g = this.RecreateDrawG() ;“
       “  g.append(‘svg:circle’).attr(‘x’, obj.fX).attr(‘y’, obj.fY);“
      “}“;
    painter->Draw(this, jsfunc);  // provide draw function directly
}
```

Graphics multithreading

- Remove all global pointers, used for graphics
  - gPad, gVirtualX, gVirtualPS
- Keep gStyle
  - make it thread-safe
  - collection of default graphics properties
- Allow to create canvases in different threads
- Canvas and drawn objects should be modified from the thread where they created

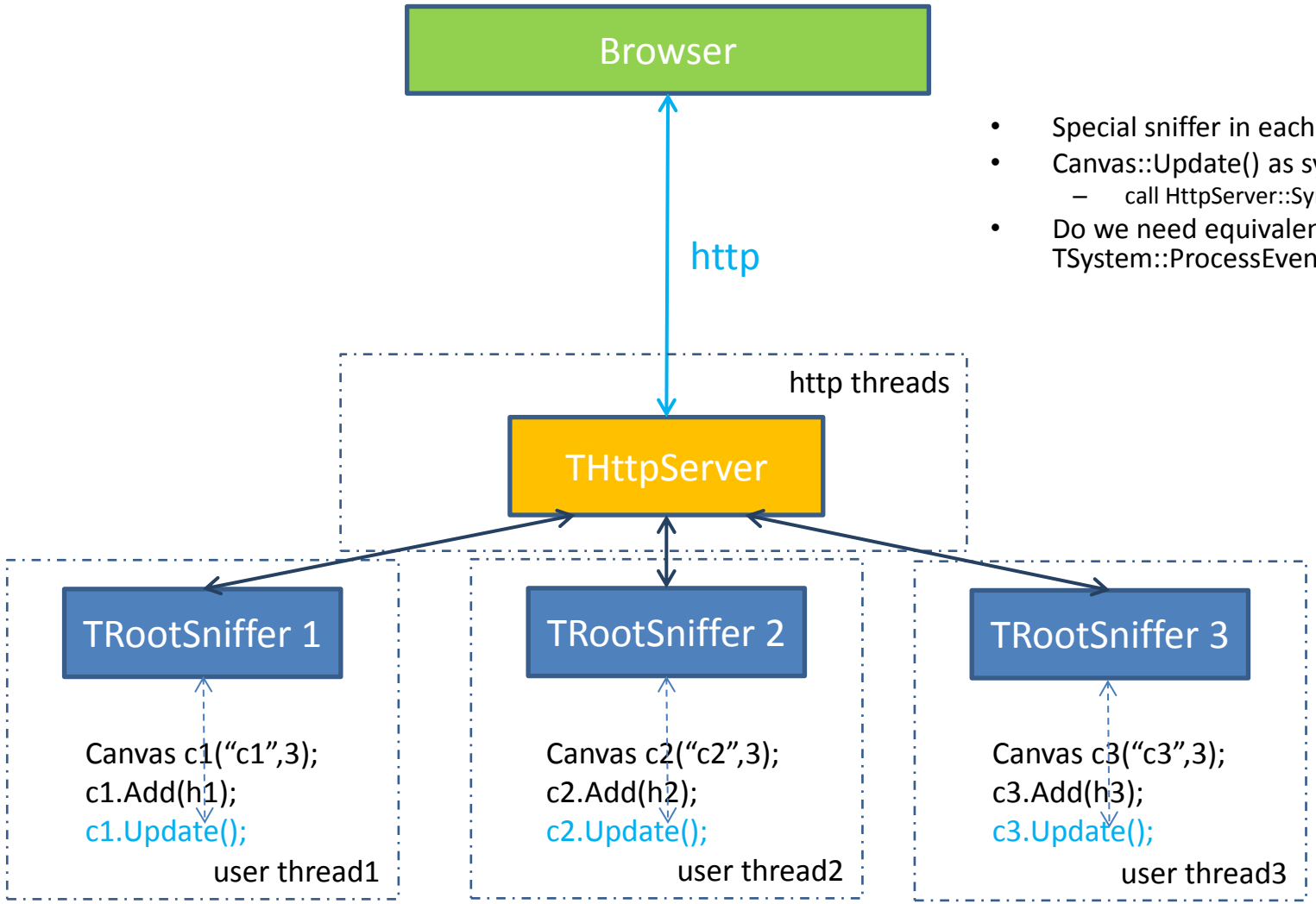
Backup slides

TCanvas with THttpServer now



- THttpServer uses own threads for sockets communication
- Sniffer interacts with objects in main thread
- Object access only within gSystem->ProcessEvents()

Many canvas in different threads



- Special sniffer in each thread
- Canvas::Update() as sync point
  - call HttpServer::Sync() inside
- Do we need equivalent for TSystem::ProcessEvents() ?

Canvas Modified() and Update()

- Canvas::Modified()
  - used to inform that all objects in canvas were changed
- Canvas::Modified(obj)
  - specify exactly object(s) which was modified
- Canvas::Update()
  - produce data for painting on client side
  - here Paint() method for every object will be called

Sync and async modes

- Now all graphics is synchronous
  - TCanvas::Update() blocked until painting is finished
- Can be done asynchronous
  - JavaScript client will perform drawing independently from server activity
  - let run server code faster
- Synchronous mode can be supported as well
  - as default?

GUI support

- GUI will be build with HTML/JavaScript frameworks
  - event handling in the browser
  - only final actions should be executed in C++
- Access to ROOT methods via http
  - already now supported by THttpServer
  - extend with use of websockets?
- Integrate ROOT graphics inside any other UI
  - straightforward within HTML
  - QWebView with Qt

Offline mode

- Display ROOT objects from the files
  - like JSROOT now
- Should be supported as well
  - just reuse same JavaScript code
- May required some code duplication
  - like stat box filling after change of zoom

ROOT6 prototype

- Subclass of TCanvasImp
- Minimal changes in base ROOT classes
- Implements all basic features
- Can be used as testbed

- Not all ROOT6 classes will be supported

ROOT6 prototype status

+ THttpServer and websocket
+ long polling as fallback
+ multiple client support
+ batch mode for image production
+ first tests with *Chromium Embedded Framework*
+ first tests with *Qt QWebView*
+ new canvas Paint() logic
+ new object Paint() logic
+ drawing and update with JSROOT