

Server/Client approach for ROOT7 graphics

Sergey Linev,

April 2017

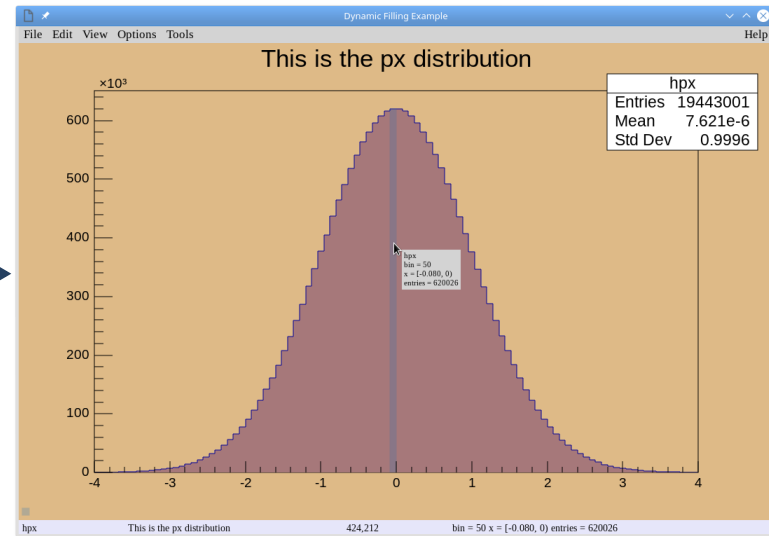
Server/Client model

- 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

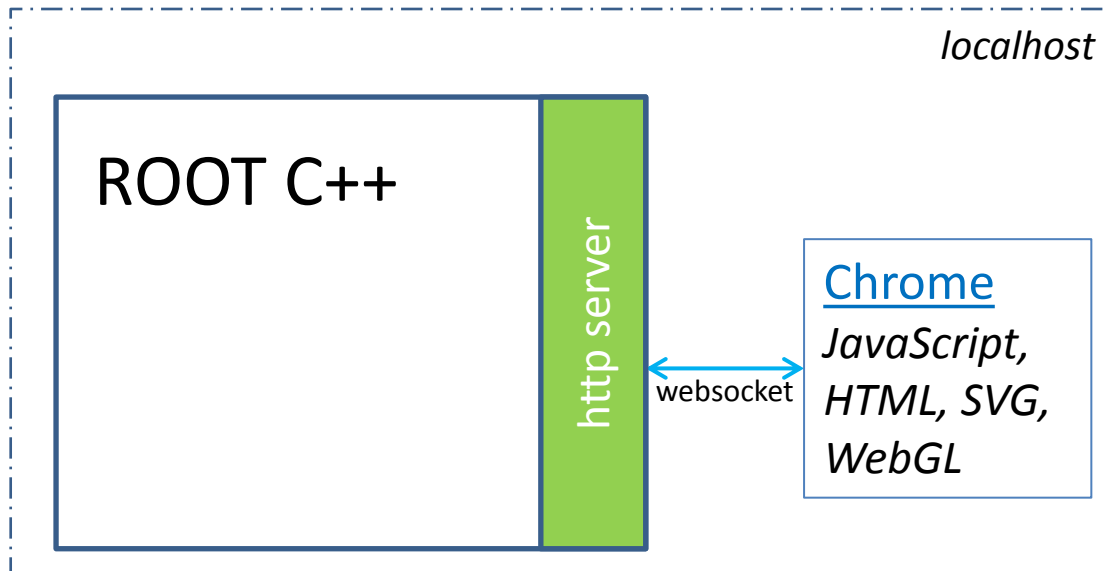
Client

any web browser

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
- Unique objects ID for primitives in pad:
 - to get list of context menu items
 - to execute objects method

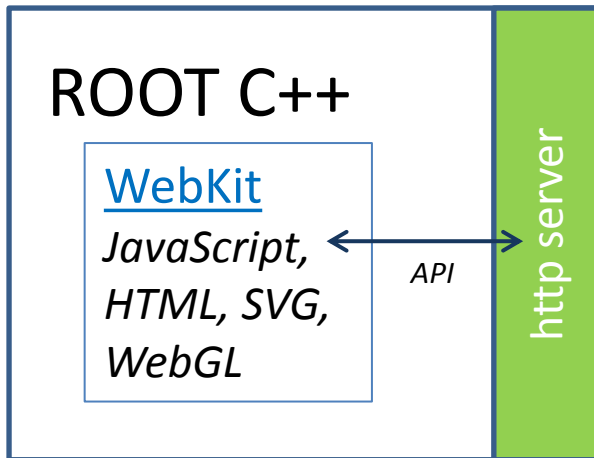
Local clients



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

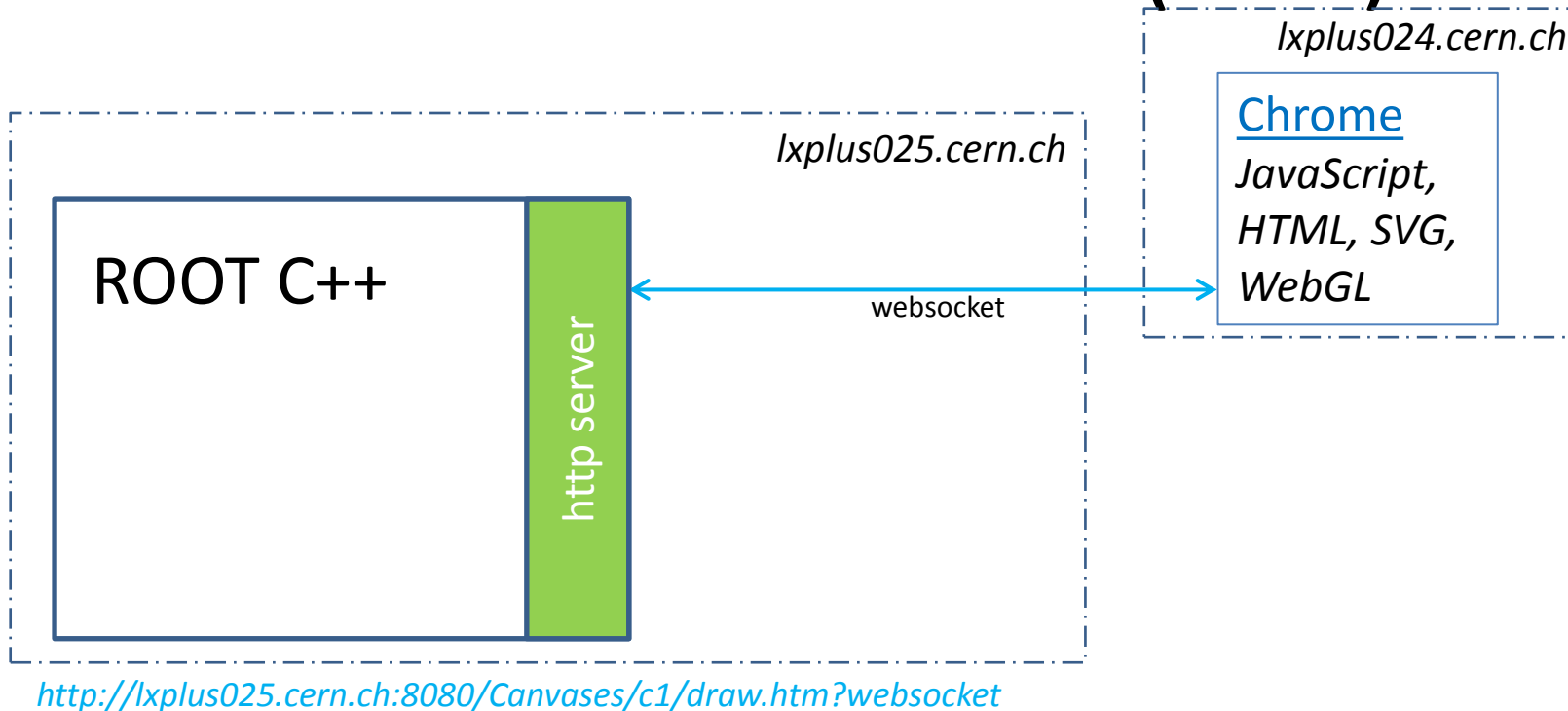
- Run any browser on the some host
- Use THttpServer bound to loopback address
- Communication via websockets

Local clients



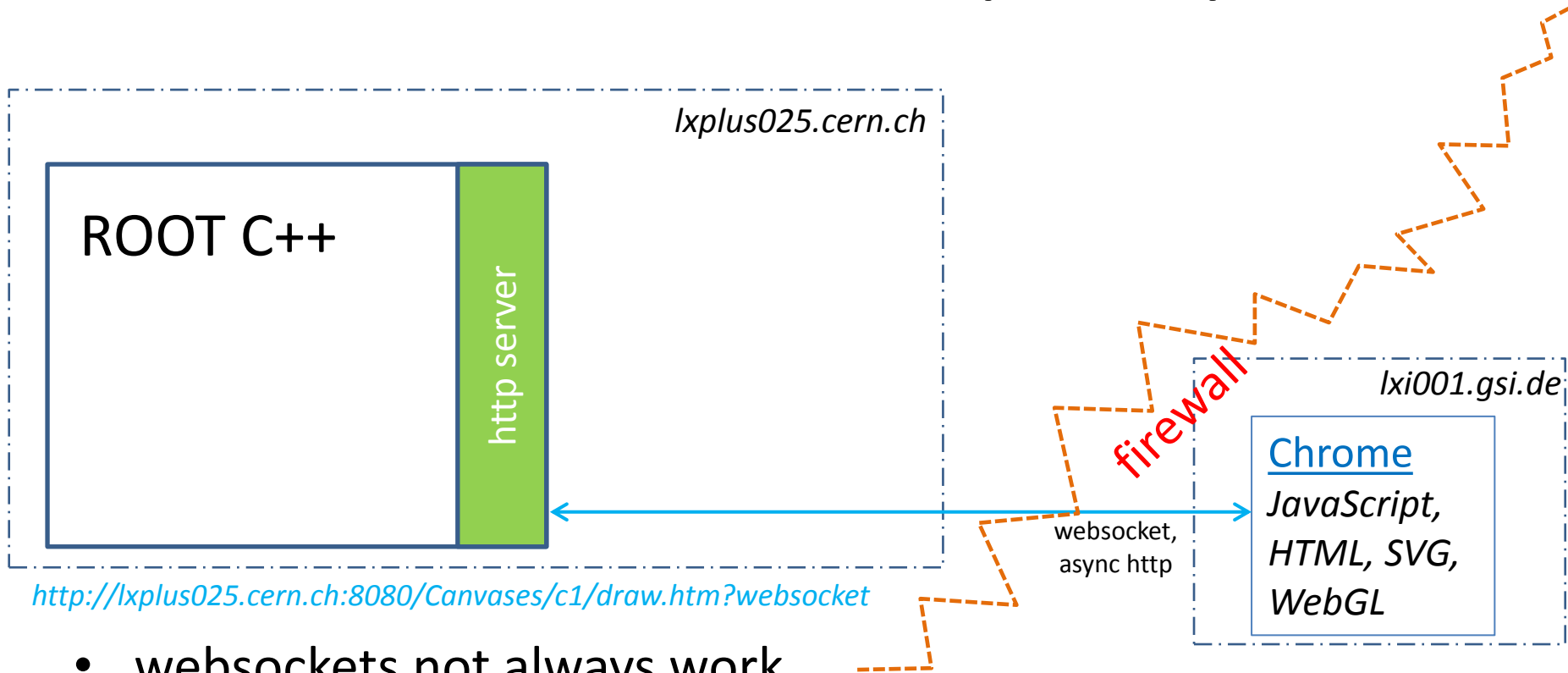
- Use WebKit or libChromium
- Create necessary window(s) directly in C++
- Communication with WebKit API (no any sockets)

Remote clients (LAN)



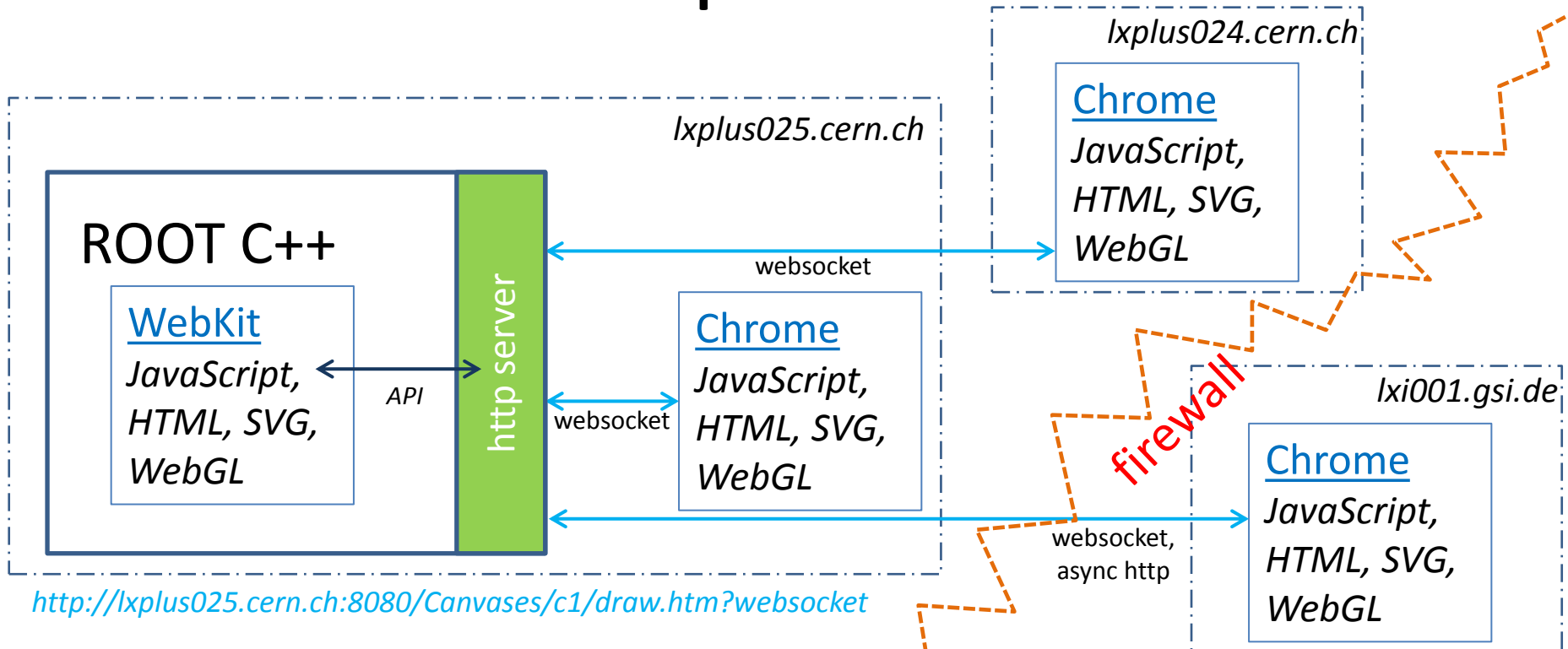
- No any difference with local clients
- Bound THttpServer with normal IP address
- Communication via websockets

Remote clients (WAN)



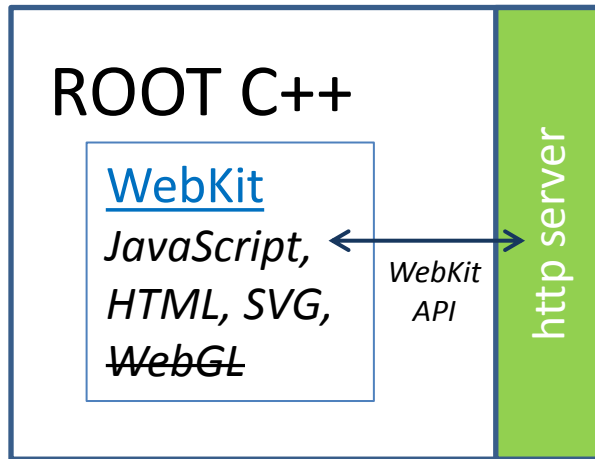
- websockets not always work
 - firewall/proxy limitation
 - not supported in `fastcgi`
 - automatic fall back to long polling (async http)

Multiple clients

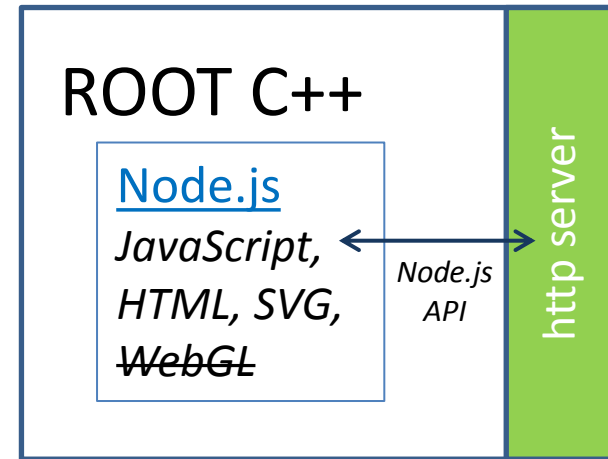


- Same canvas can be displayed multiple times
- Separate websocket for each client
- Zoom in any client propagated to all others
 - clients are stateless

Batch mode



or



- 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
}
```

Draft, subject to change

Paint examples

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

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

Draft, subject to change

Paint examples

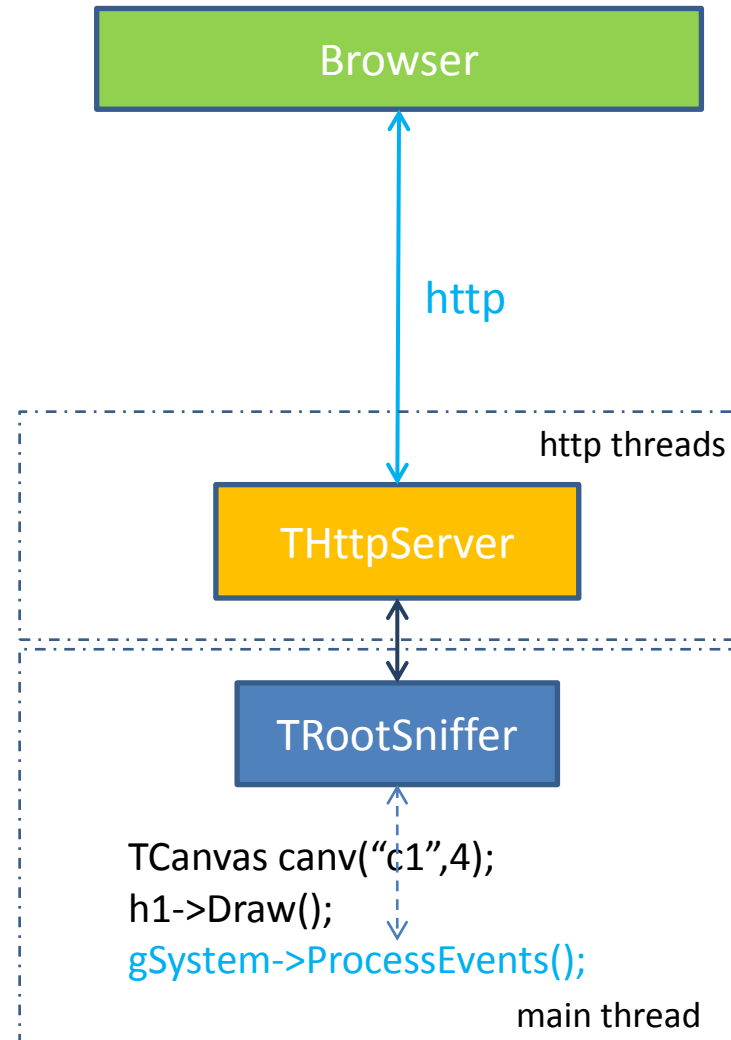
```
void UClass::Paint(PadPainter* painter)
{
    std::string jsfunc =
        "function(obj) {"
        "  var g = this.RecreateDrawG() ;"
        "  g.append('svg:circle').attr('x', obj.fX).attr('y', obj.fY);"
        "}";
    painter->Draw(this, jsfunc); // provide draw function directly
}
```

Draft, subject to change

Graphics multithreading

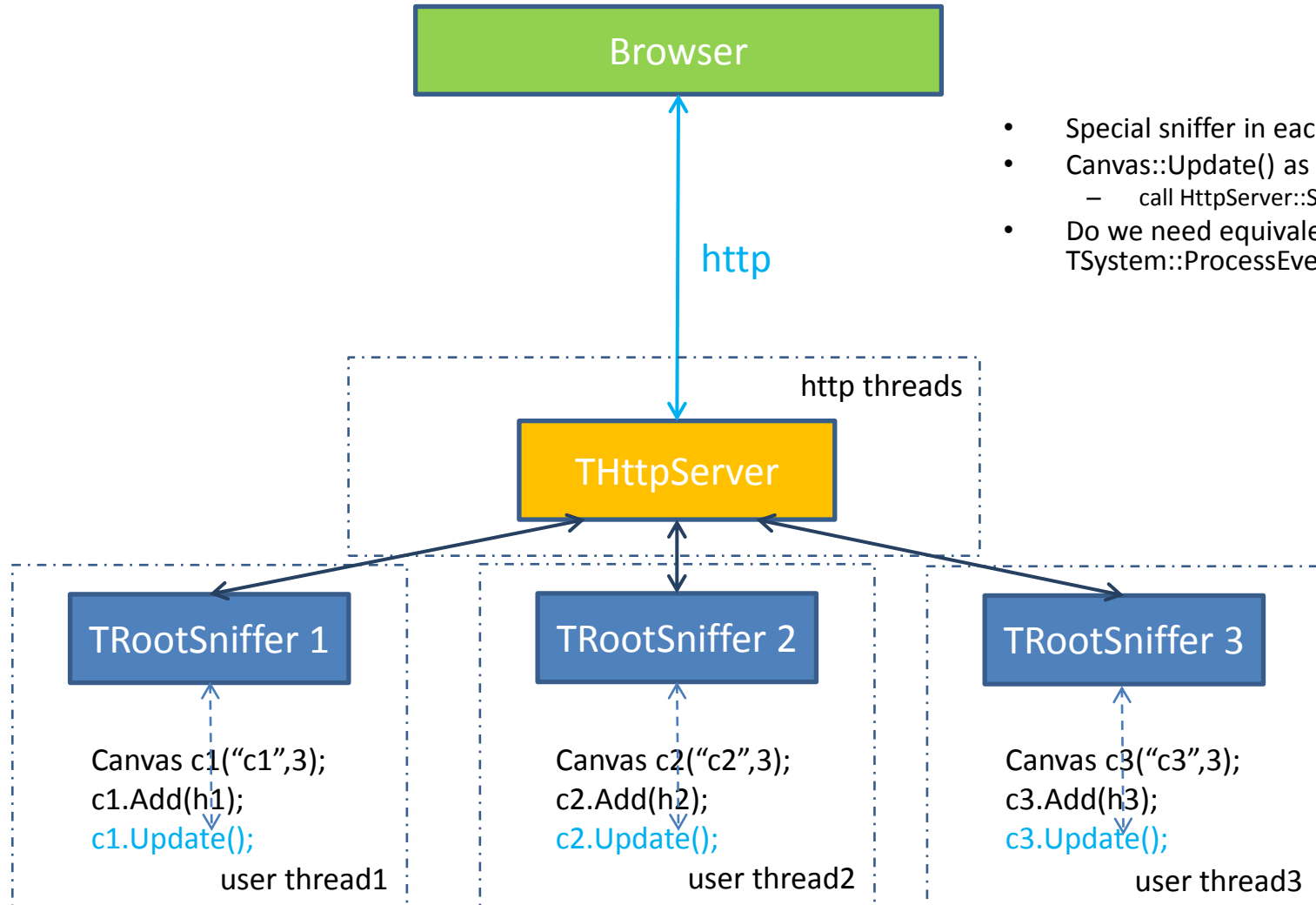
- Remove all global pointers, used for graphics
 - gPad, gVirtualX, gVirtualPS
- Probably, only keep **gStyle** and **gROOT**
 - make them thread-safe
- Allow to create canvas in different threads
- **IMPORTANT!** Each canvas and drawn objects should be modified from the thread where they created

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() ?

Backup slides

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 Qt QWebView
- + new canvas Paint() logic
- + new object Paint() logic
- + drawing and update with JSROOT