Intro to WORKERS For webRTC

Tim Panton

Worker Interface

- Background task
- Off main thread
- In JavaScript
- No access to DOM
- Uses PostMessage to communicate
- Limited access to APIs

Variants

- Shared Workers - accessed by multiple (same origin) pages

- Service Workers - can intercept fetch() and access page Cache

- Server Side - Cloudflare offers service worker API on their CDN edge - (cf streaming)

Worklets

- Workers that run on a specific browser thread
- Have own APIs

Paint Worklet	Custom CSS	Main Thread
Audio Worklet	Custom Audio	Audio render Thread
Animation Worklet	Procedural Animations	Compositor Thread

Lifecycle

	Web Workers	Service Workers
Tab	Many per tab	One for all Tabs
Lifespan	Same as Tab	Independent
Good for	Parallelism	Offline

Relevance to WebRTC

- Recording, monitoring post processing media streams
    - Silence removal from podcast recordings
    - Async notification of barcodes in video (warehouse)
    - Sharing of media across multiple pages without dropping a call (sans iframe)
    - Answerphone :-)

Data Channel

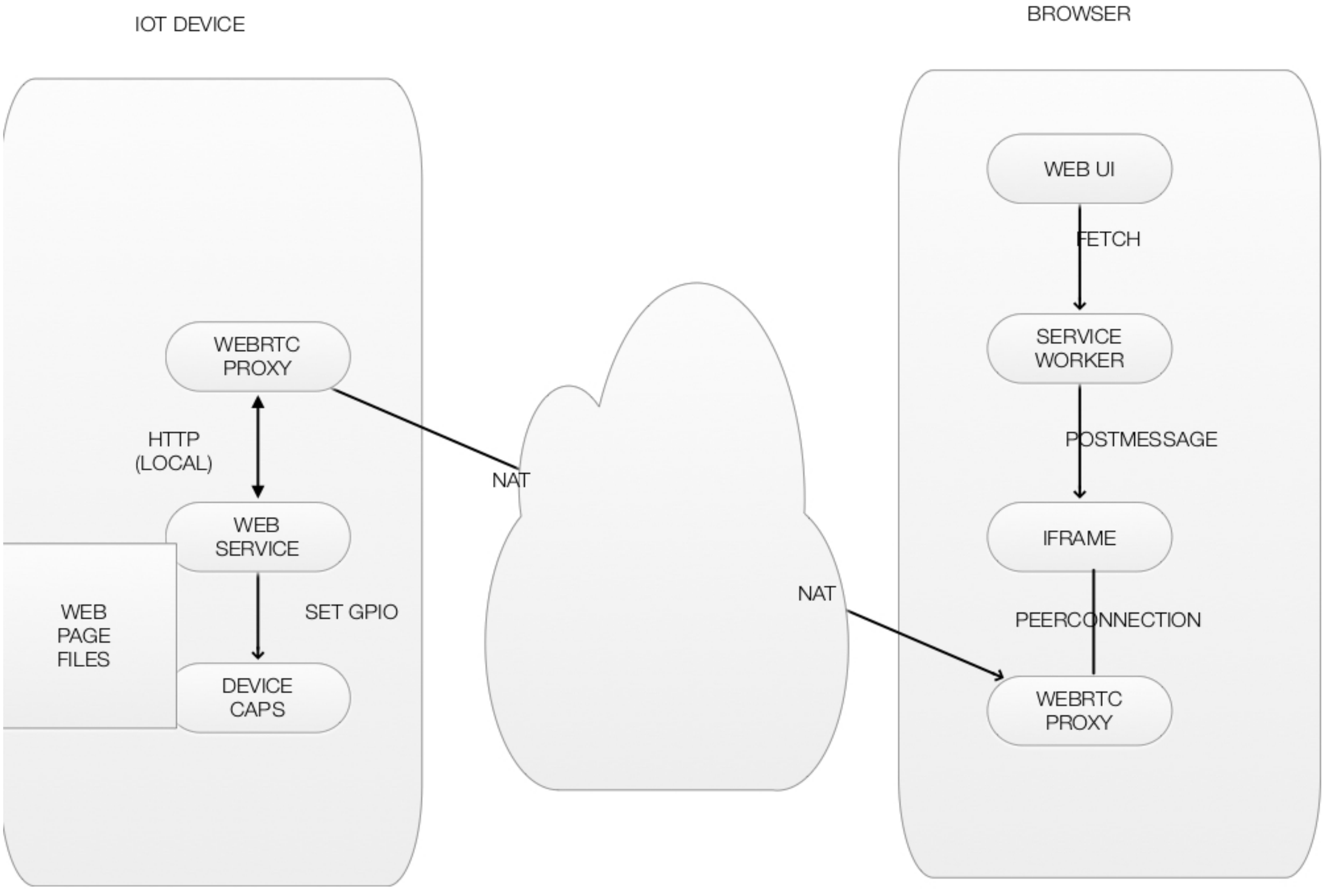
- Collect data for use by multiple pages (like the audio example)

- Serve pages from behind NAT over the data channel

DEMO

WHAT YOU SAW

- Web app
- On small device
- Behind NAT
- Rendered to a smartphone browser
- Also behind NAT


IOT DEVICE
WEBRTC
PROXY
HTTP
(LOCAL)
WEB
SERVICE
WEB
PAGE
FILES
SET GPIO
DEVICE
CAPS
NAT
NAT
BROWSER
WEB UI
FETCH
SERVICE
WORKER
POSTMESSAGE
IFRAME
PEERCONNECTION
WEBRTC
PROXY

Benefits

- Low latency
- E2E encrypted
- WebPage and service not changed
- Dynamic device pages simple
- webRTC security promises

Conclusions

- We can leverage workers to solve some of our other use cases
- Need:
  - API access to webRTC in workers
- Transferable:
  - peerconnection
  - datachannel
  - media streams