Mindroid

Crafting Software Platforms

What is the Mindroid Platform?

- **Component-based application framework similar to Google‘s Android**
- **Actor model as core system architecture building block**
- **Event-based programming paradigm (no cyclic programming paradigm)**
- **Distributed system of cooperating services (distribution transparency)**
- **Software Development Kit (SDK) including documentation and examples**
- **Heavily influenced by Android (Android API compliance), Erlang, Alan Kay (real OOP), Reactive Manifesto**
- Resource management
- Resource monitoring
- Package management (apps)
- Security (Android-style permissions)
- Service Discovery
- Logging and bug reports
- Integration test framework and mocking environment
- Mindroid core is open-source (Apache 2 License)
- Mindset and culture

Why Mindroid?

- Modularity: Components with clear interfaces, threading and dependencies
- Reuse: Set of reusable components across projects and platforms
- SDK: Sustainable, public APIs crafted by an API first design approach
  → Finding good abstractions and truly care about naming things
- Testability and refactorings: No shared state between components
- Reliability by Erlang-style component supervision
- Software quality highly benefits from Actor model design approach
  → No complex critical sections
  → Callbacks run in right thread contexts
  → Low energy, CPU time and memory requirements
- Logging (No long-lasting debugging sessions)
- Slim platform
  → Simplicity is prerequisite for reliability (Edsger W. Dijkstra)
- Scalability: Distributed system of cooperating services running multiple Mindroid instances on different nodes
- Time to market

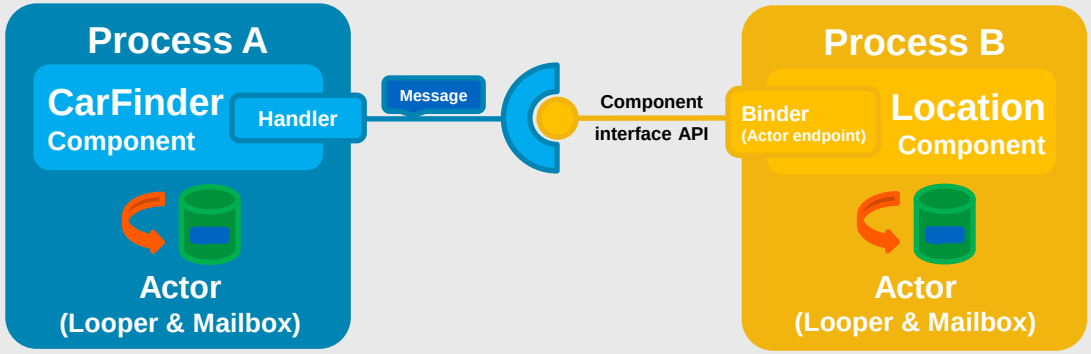
Mindroid Actor Model

Erlang-style Actor Process Model

1. Component isolation: No shared state across component boundaries
   - Component interfaces implemented using actor endpoints
   - Component interfaces exchange only POD types and component interfaces (actor endpoints) to access further components
   - No complex critical sections
2. Deployable components within lightweight processes
   - Modularity
   - Architectural flexibility
3. Asynchronous message-passing
4. Mailboxes buffer incoming messages
5. Component interface API wraps the message passing paradigm

Alan Kay: *“The big idea is ‘messaging’. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.”*

Mindroid Actor Model by Example




```
1  public class CarFinder extends Service {
2      private LocationManager mLocationManager;
3
4      public void onCreate() {
5          mLocationManager = new LocationManager(this);
6          mLocationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 1000, 10, mLocationListener);
7          Location location = mLocationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
8          Log.d("CarFinder", "Last known location: " + location);
9      }
10
11     public void onDestroy() {
12         mLocationManager.removeUpdates(mLocationListener);
13     }
14
15     public IBinder onBind(Intent intent) {
16         return null;
17     }
18
19     private LocationListener mLocationListener = new LocationListener() {
20         public void onLocationChanged(Location location) {
21             Log.d("CarFinder", "Location changed: " + location);
22         }
23     };
24 }
```

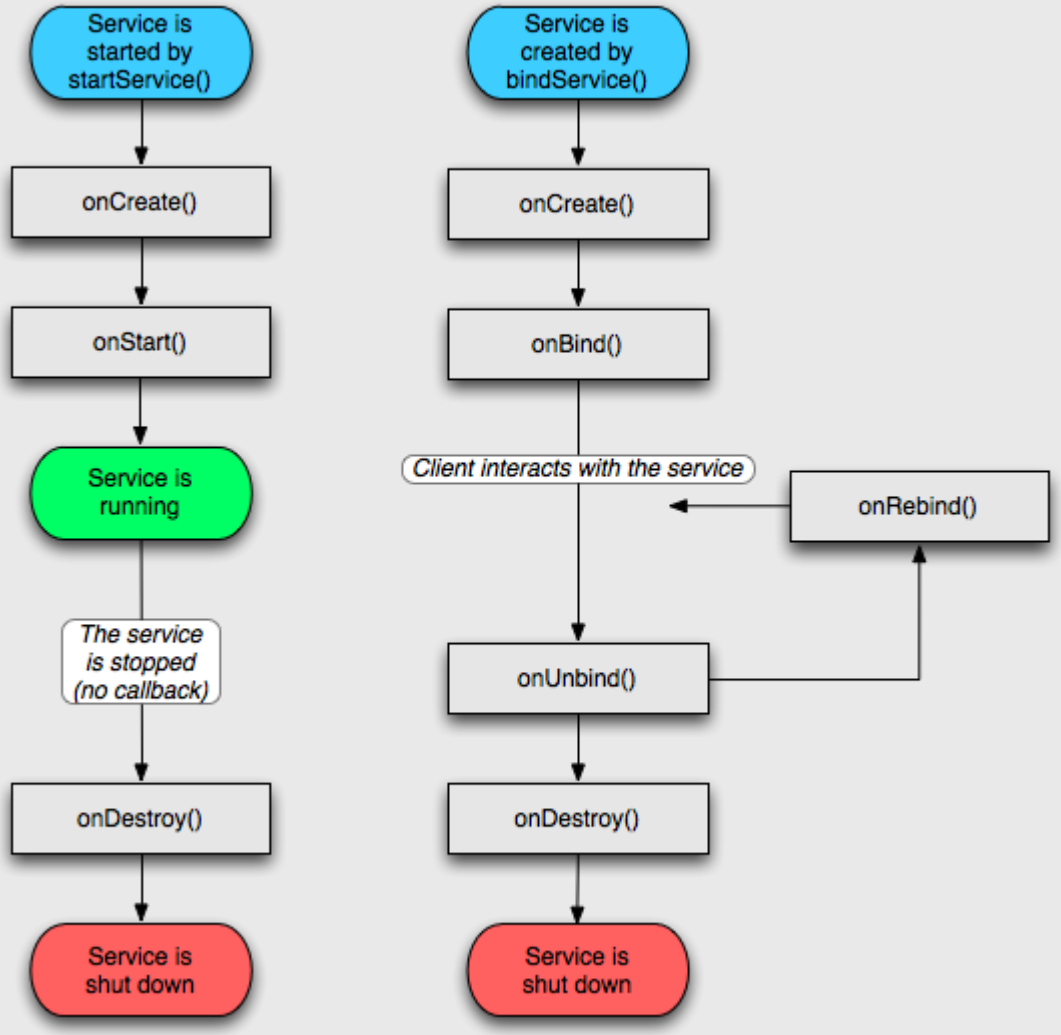
Mindroid Variants

1. Mindroid.java: Targeting the Java platform
2. Mindroid.cpp: Targeting native platforms, e.g. Linux or QNX Neutrino RTOS
   - Modern C++ using Java-style collections and reference counting to improve security and to avoid memory leaks
3. Mindroid.ecpp: Targeting deeply embedded systems without dynamic memory
   - Reduced feature set building upon the event-based actor model programming paradigm
   - e.g. AUTOSAR SWC, AUTOSAR OS, CMSIS RTOS, bare metal, and POSIX environment for testing

Mindroid Services

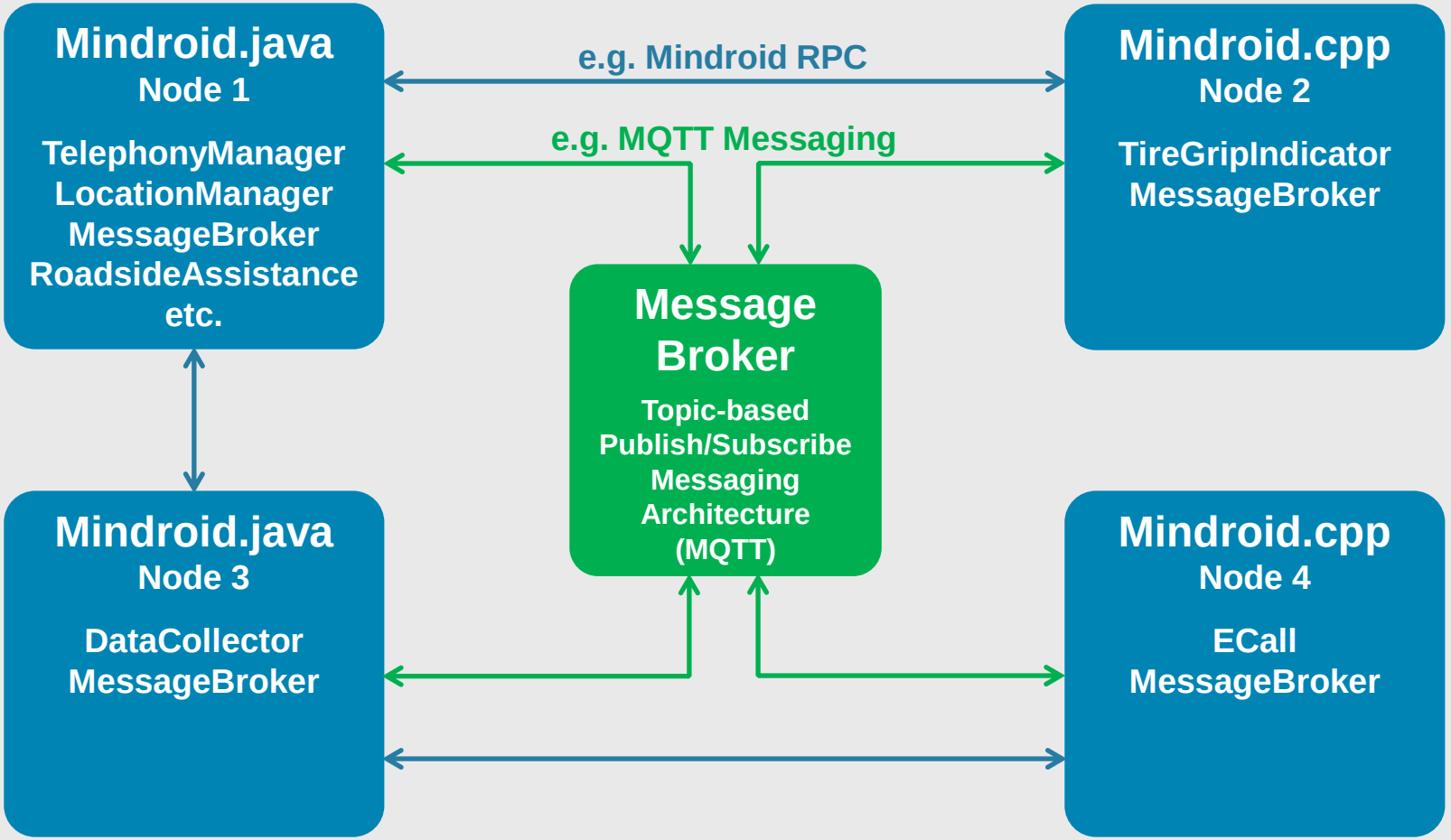
- MessageBroker
  Extensible, QoS-aware, topic-based publish/subscribe message broker currently supporting MQTT, REST and SOME/IP using various domain model formats like XML, EXI, JSON, ProtoBuf, SOME/IP
- PowerManager (Wake locks and leases)
- AlarmManager
- LocationManager
- MediaPlayer
- AudioManager
- TelephonyManager
- Logging
- PackageManager
- ConnectionManager
- SupervisionService
- DownloadManager
- OTA-UpdateManager
- Diagnostics
- …

Mindroid Service Lifecycle



Distributed Mindroid

Example Multi-Process, Multi-Device, Multi-Domain Architecture



Erlang Design Principles

- The world is concurrent
- Things in the world don't share data
- Things communicate with messages
- Things fail

Automotive ConnectedCar Device

A real-world device out there in many cars

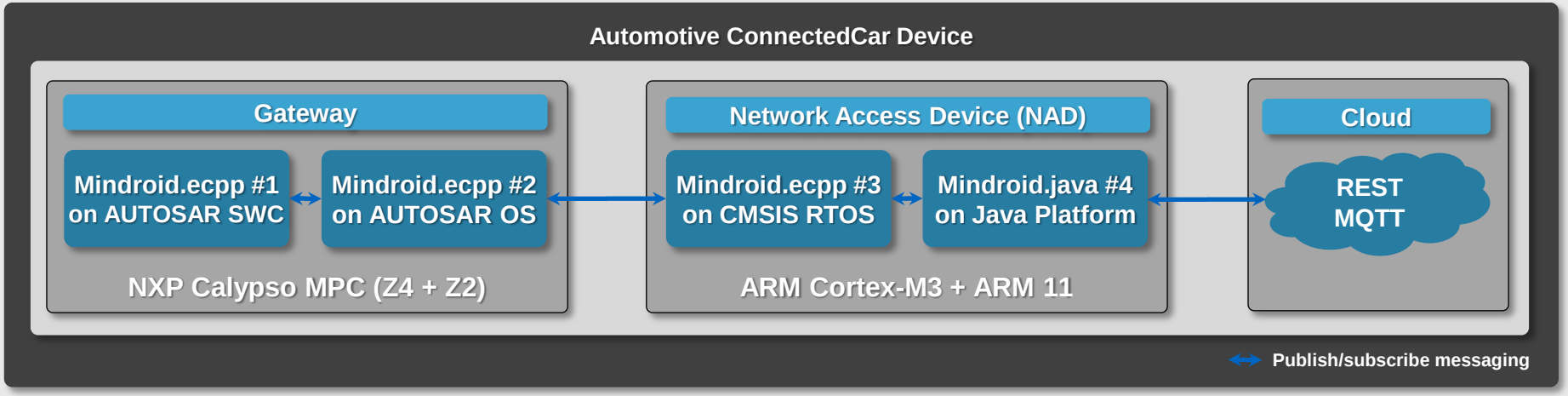
Automotive ConnectedCar Device

Building Blocks

1. Mindroid application frameworks
2. Topic-based publish/subscribe messaging
   - Most vehicle communication perfectly fits the publish/subscribe messaging paradigm
3. Apps

Hint: Mindroid actor model message passing != publish/subscribe messaging

Automotive ConnectedCar device runs 4 Mindroid instances



Lessons Learned I

- Always start small with a small team and common sense
- Focus on solving the hardest problems first
- Competitive situation fuels motivation → be fast(er), keep your work honest
- Truly care about things (the project)
  → act, push things forward
  → listen to customer feedback
  → do not avoid discussions with the customer
- Gain customer trust by continuous high performance
- Failures happen; be fast in analyzing, finding and fixing them → customer trust
- Open communication within the team → push system, no pull system
- Know about (software) technologies, not just programming languages
- Phone calls → talk to each other directly, not only via emails or tickets
- Logging and logging guidelines → quality indicator

Lessons Learned II

- Keep an eye out for beauty and aesthetics in your code base no matter if it is about architecture, naming, etc. → it is not just about getting work done
- Code reviews → read every checkin
- Code maintainability → favor fast code reading, not fast code writing
- Reduce risk by integrating early and often
- Simplicity is prerequisite to reliablility
- Continuously monitor project progress
- Do engineering, not tinkering (gothic cathedral vs. dog house)
- Find the right building blocks (abstractions) to get simplicity → design process
- Solve the whole problem context, not just a problem → building blocks
- Keep basic interfaces (public API) stable → Butler Lampson

Lessons Learned III

- Automotive ConnectedCar device as personal research project to try out ideas
  → Alan Kay: Real OOP with messaging as the big idea
  → Carl Hewitt, Tony Hoare: Actor model and CSP
  → Joe Armstrong: Erlang-style actor process model
  → Andrew Tanenbaum: Distributed system design, modular architecture
- Be careful with “endless” passion in automotive series projects