



Multiuser

1	Contents	
2	Terminology and concepts	3
3	“user”vs. “uid”	3
4	Trusted components	4
5	System services	4
6	User services	4
7	Multi-seat (logind seats)	5
8	Fast user switching	5
9	Requirements	5
10	Distinguishing between privacy levels in user-specific data	7
11	Authentication	8
12	General use-cases	8
13	First use	8
14	Individual use: preferences and state restored	8
15	User switching	9
16	Guest mode	9
17	Borrowing the car	10
18	Existing multi-user models	10
19	Switchable profiles without privacy	10
20	Typical desktop multi-user	11
21	Basic multi-user: log out, log in as another user	11
22	Fast user switching: switching user without logging out	12
23	Multi-user desktops with multi-seat support	13
24	Android 4.2+	13
25	Multi-user support in the Tizen 3 automotive platform	14
26	Approach	16
27	The principle of least-astonishment	16
28	Levels of protection between users	16
29	User accounts: representing users within the system	17
30	Sharing one uid between all users	17
31	One uid per user	18
32	Multiple uids per user	18
33	Creating and managing user accounts	19
34	Registering the users	19
35	The first user to be registered is special	20
36	Graphical user interface and input	20
37	Single compositor	21
38	Switching between compositors	21
39	Switching between compositors with a system compositor	22
40	Switching between users	23
41	Preserving “core”functionality across user-switching	24
42	System services	25

43	User services continuing to run	26
44	Distinguishing between the driver and other users	26
45	Agents	26
46	Returning to previous state	26
47	Application ownership and installation	27

48 **Summary of recommendations** **27**

49 This document describes how multiple users are expected to use the Apertis
50 system, and works mostly as a guide and recommendations to help designing
51 the system. It is intended to act as an “umbrella” document covering the multi-
52 user topic in general, and will be supplemented by more concrete documents
53 describing particular use-cases and recommendations for how those use-cases
54 can be addressed.

55 The driving force behind having a multi-user system is to allow customization
56 of the system. A system may have multiple users who would be frustrated by
57 customizations done by each other to the system’s look and feel and even to
58 data such as playlists. Having multiple users allows each to customize their own
59 interface.

60 Depending on OEM and consumer requirements, multi-user systems can poten-
61 tially also provide personal files and online accounts for each user.

62 **Terminology and concepts**

63 **“user”vs. “uid”**

64 In a Unix system, users are typically identified by a numeric user ID, often
65 abbreviated “uid”. A uid can represent a person, a system facility, multiple
66 people, or even an application (as in Android).

67 Because these do not correspond 1:1 in some designs, it is important to be clear
68 which one is under discussion. In this document, the jargon term *uid* or *user*
69 *ID* is used to refer to a Unix user identifier, while *user* or *person* is used to refer
70 to a human using the system.

71 *User account* refers to any abstract representation of the user within the system.
72 This is most commonly a uid, matching the original Unix design. However,
73 systems can exist with multiple uids per user account, such as Android, in
74 which each (user account, app) pair has a uid. Conversely, systems can exist
75 with multiple user accounts sharing a uid, such as SteamOS (in which one uid
76 runs the Steam Big Picture UI, and users log in to it with separate Steam
77 accounts).

78 The canonical form of a Unix uid is numeric, but for ease of reference, a short
79 lower-case textual *username* may be used to refer to a uid. For example, it
80 is common to talk about system users named “root”and “backup”, but the real

81 identities of these users within the system are the corresponding numeric uids
82 0 and 34; the usernames are merely for convenience and mnemonic value.

83 **Trusted components**

84 A *trusted* component is a component that is technically able to violate the
85 security model (i.e. it is relied on to enforce a privilege boundary), such that
86 errors or malicious actions in that component could undermine the security
87 model. The *trusted computing base* is the set of trusted components. This is
88 independent of its quality of implementation –it is a property of whether the
89 component is relied on in practice, and not a property of whether the component
90 is *trustworthy*, i.e. safe to rely on. For a system to be secure, it is necessary
91 that all of its trusted components must be trustworthy.

92 One subtlety of Apertis’app-centric design is that there is a trust boundary
93 between applications even within the context of one user. As a result, a multi-
94 user design has two main layers in its security model: system-level security that
95 protects users from each other, and user-level security that protects a user’s
96 apps from each other. Where we need to distinguish between those layers, we
97 will refer to the *TCB for security between users* or the *TCB for security between*
98 *apps* respectively.

99 **System services**

100 A *system service* is a service that, conceptually, runs on behalf of the whole
101 computer, without a division between users. In designs where each user has
102 a distinct uid, system services run under a system uid, either root (the most
103 privileged uid) or a special unprivileged uid per service or group of services; they
104 do not run with the uid of any particular user.

105 This term does not necessarily imply anything about whether the service is
106 considered to be “part of the operating system”, or whether it is part of a pre-
107 installed or user-installable application bundle as discussed in the Applications
108 Design document. However, because system services can accept requests from
109 multiple users, any system service that will handle users’private data must be
110 trusted to impose a privilege boundary.

111 Examples of system services commonly present in Linux systems include Conn-
112 Man, NetworkManager, BlueZ, udisks and the D-Bus system bus.

113 **User services**

114 A *user service* is a service that runs on behalf of a particular user. In designs
115 where each user has a distinct uid, each user’s user services typically run under
116 that same uid; in designs like SteamOS where all users share a single generic
117 uid representing “all users”, user services would typically share that same uid.

118 Examples of user services commonly present in Linux systems include dconf,
119 gvfs, Tracker, Tumbler and the D-Bus session bus.

120 Identifying a process as a user service is independent of whether it is treated
121 as part of the Apertis platform and independent of any particular application
122 (such as the user services mentioned above), or treated as part of an application
123 bundle (“agents” associated with apps).

124 Multi-seat (logind seats)

125 In the context of a multi-user system, a *seat* is a collection of display and input
126 devices, optionally linked to other devices such as a USB socket or optical drive,
127 intended to be used by one user at a time. Typical PCs only offer one seat,
128 but a second graphics adapter, often connected via USB, can be used to add
129 additional seats (a *multi-seat* system).

130 This jargon term is commonly used in Linux system services such as systemd-
131 logind, the older ConsoleKit, and GDM. As an example, in the context of a
132 car, it should be noted that it does not necessarily correspond precisely to the
133 car’s seats: for instance, in the common layout that places a single “head unit”
134 touchscreen between the driver and front passenger, that touchscreen and any
135 USB sockets adjacent to it would be treated as a single seat. If, for example,
136 additional touchscreens were added behind the front seats for use by rear pas-
137 sengers, that would be a multi-seat system with 3 seats (front, rear left, rear
138 right).

139 Apertis uses systemd-logind as a core system service, so where disambiguation
140 is needed, we will refer to this as a *logind seat*.

141 Fast user switching

142 Many operating systems have the concept of *fast user switching*, which is de-
143 scribed in **Fast user switching: switching user without logging out**. Following
144 common usage, this document reserves the term “fast user switching” to refer to
145 that particular multi-user model, even if some other model might be equally
146 fast or faster in practice.

147 Requirements

148 Apertis is currently designed as a single-user system. There is one GUI session
149 with full access to all preferences, apps and data, and a set of apps and user
150 services with varying levels of sandboxing and privilege separation from each
151 other within that session, running on top of system services whose privileges vary.
152 The high-level requirement for this document is that this should be expanded to
153 support multiple GUI users, each with their own private data and user services,
154 running on top of similar system services.

155 This section contains a list of general requirements applicable to many multi-user
156 systems.

- 157 • Multiple users should be able to use the system. Depending on the specific
158 set of requirements, this could involve concurrent use, or one user at a time.
- 159 • When the user logs in to a newly started system, they should find the
160 same applications they had left open last time they shut down the system,
161 and in the same state. See [Returning to previous state](#) for discussion of
162 this topic.
- 163 • Some data is private to each user. Depending on the specific set of re-
164 quirements, this could include:
 - 165 – Settings
 - 166 – Address book
 - 167 – Browser history
 - 168 – Application icons
 - 169 – Arrangement of icons in the app launcher
 - 170 – Account data for web services
 - 171 – Playlists
- 172 • Some data is shared between users. Depending on the specific set of
173 requirements, this could include:
 - 174 – Applications (from the store)
 - 175 – Media library (music, videos)
- 176 • Depending on the specific set of requirements, switching users at runtime
177 could be supported. Where it exists, this shall be performed with a smooth
178 transition, with no visual flickering. User switching should not take more
179 than 5 seconds. See [\[Switching between users\]](#) for discussion of this topic.
- 180 • A subset of features are considered to be core functionality, and must
181 not be disturbed by switching between users: they must remain available
182 before, during and after any transition between users. The set of core
183 functionality could vary by device; in this document we mainly use music
184 playing and navigation as examples of this category. See [Preserving “core”](#)
185 [functionality across user-switching](#) for further discussion of this topic.
- 186 • The subset of features that are not disturbed while switching between
187 users must not be limited to functionality that is considered to be “part of
188 the operating system”. For example, it should be possible to place a user-
189 installable player for a third-party music streaming service such as Spotify
190 or last.fm in this category. Again, see [Preserving “core” functionality across](#)
191 [user-switching](#)

- Depending on the specific set of requirements, peripheral hardware devices such as USB storage devices and paired Bluetooth devices could either be shared across the entire system, or specific to a user. If they are shared, then they must be accessible to all users, with all users able to unmount/eject them.
- The authentication and user-switching user interface should not distract the driver more than is necessary; for instance, they should not ask security or authentication questions unless a decision is strictly required.
- The user privileges of the system should be visually obvious: if users have selected different personalizations such as colour schemes or themes, then the display should use a particular user's theme whenever it is acting on behalf of that user, and at no other time. This limits the risk that users will encounter undesired privacy consequences resulting from misunderstanding the system's privacy model.

Distinguishing between privacy levels in user-specific data

There are several possible categories of user-specific data.

Some user-specific data is private. For instance this might include email, browsing history, social media feeds. (Alice should not be able to read Bob's email, history, social media feeds and so on unless Bob has allowed it.) Meanwhile, some user-specific data is sensitive because it allows acting on someone else's behalf. (If Alice is logged-in to Amazon, Bob should not be able to buy things using her account.) Private and sensitive data are interchangeable from a systems perspective: they must be accessible by that user, and only by that user.

However, some data is only user-specific for convenience or organization; it isn't important whether other users are able to read it, as long as it doesn't make their own actions less convenient.

For instance, the set of apps that are visible in menus might be one example of user-specific data that does not necessarily need to be treated as private. If Alice has installed apps for social media networks that Bob doesn't use, they shouldn't appear in Bob's menus —but if Bob specifically looks for them, perhaps in an Android-Settings-like “storage usage” view, it might be considered acceptable that he can see what Alice installed.

Another possibility for sharing data is that playlists within a shared media library could appear as an unobtrusive “Bob's playlists” folder in other users' menus, if desired.

As discussed in [Levels of protection between users](#), the level of privacy and integrity protection between users can vary according to OEM and consumer requirements; this could influence how user-specific data is categorized.

230 Authentication

231 We assume that the HMI provides a way for users to identify and authenticate
232 themselves to a trusted HMI component, for instance by:

- 233 • presence of a unique physical key
- 234 • presence of a personal item such as a phone with Near-Field Communica-
235 tion support
- 236 • a password or lock-screen gesture
- 237 • face or fingerprint recognition
- 238 • simply selecting a user from a menu (choice of user, but no meaningful
239 authentication, similar to one of the cases described in [Switchable profiles
240 without privacy]

241 The exact authentication mechanism depends on manufacturer and user require-
242 ments, and is outside the scope of this document: this document only assumes
243 that an identification/authentication mechanism exists as part of the operating
244 system, and does not rely on specific properties of that mechanism.

245 General use-cases

246 While this document does not go into the specifics of more elaborate use-cases,
247 there are a few simpler use-cases which should be considered by any concrete
248 multi-user design within the framework established by this document. In some
249 cases these use-cases could be considered and rejected, if a particular design's
250 requirements put them out of scope. To show them, the context of a car will be
251 used as an example.

252 First use

253 Alice uses the car for the first time. The system recognizes that she has not
254 used it previously and so there is no saved state.

255 **a. First use:** The system starts in some default state, for instance at a main
256 menu or with a default application such as a media player running.

257 Individual use: preferences and state restored

258 Alice and Bob share a car, and have separate keys. Alice has configured the
259 display for a red UI theme; she uses the car on Monday, listens to a podcast while
260 she drives, and has the email app open in the background. Bob has configured
261 the UI for a blue theme. He uses the car on Tuesday, and reads the BBC News
262 website in the browser app while stopped at motorway services.

263 **a. Last-used mode:** The next time Alice starts the car and authenticates
264 as herself (see [Authentication](#)), the podcast and email apps should resume in
265 the same state they were in when she shut the system down on Monday, and

266 the HMI configuration should reflect her preferences (the red theme should be
267 used, etc.). Similarly, the next time Bob authenticates as himself, the BBC
268 News website should be displayed in the browser app as it was when he shut
269 the system down on Tuesday, and the blue theme should be used.

270 **b. Privacy between non-concurrent users:** If the system is configured
271 to provide protection between users, then Alice’s private data should not be
272 available to Bob and vice versa. For instance, Bob’s web browsing history and
273 social media accounts should not be available when Alice starts the web browser,
274 even if Alice deliberately looks for them.

275 User switching

276 **a. User switching:** Bob is currently using the HMI to read Twitter, and Alice
277 wants to check her email. Neither is currently driving. Alice should be able
278 to authenticate in some way (see **Authentication**), switching the HMI to have
279 Alice as its current user. When she has finished, Bob should be able to switch
280 the HMI back so he is the current user again, and continue to read Twitter.

281 **b. Privacy during user switching:** after switching from Bob’s user account
282 to Alice’s, Bob should be able to go away, knowing that Alice cannot access
283 his Twitter feed. When Alice has finished and hands back control to Bob, she
284 should be able to know that Bob cannot access her email.

285 In existing multi-user systems like those described in section 4, this
286 is typically implemented by leaving Bob’s user account in a “locked”
287 state after he transfers control to Alice, and vice versa, requiring
288 re-authentication before resuming use.

289 Guest mode

290 Greg, a guest, is in Diana’s car.

291 **a. Unauthenticated guest session:** If Diana has enabled it (or if it is enabled
292 by default and Diana has not disabled it), Greg should be able to start a guest
293 session that can access public information and the Web, play music from the
294 car’s music library, etc. without authentication.

295 **b. Owner’s privacy:** Greg should not be able to access Diana’s private data
296 (or the private data of any other user of the system).

297 **c. Guest’s privacy:** Greg’s browser history, Facebook authentication token,
298 etc. should not be available to subsequent guests. For instance, the system
299 could temporarily allocate space for Greg’s user-specific data, then discard it
300 and terminate all guest processes as soon as Greg logs out, returning to default
301 settings for the next guest.

302 **d. Guest is restricted:** Greg should not be able to add or delete music, install
303 or remove apps, or similar actions.

304 **Borrowing the car**

305 Diana lends her car to David, giving him her key.

306 If the system is configured to consider a key as sufficient authentication for a
307 user, then it cannot be expected to protect Diana from malicious action by
308 David. However, if the system is configured to require secondary authentication
309 such as a password, PIN or lock-screen swipe pattern, then David will not be
310 able to use Diana's account.

311 **a. Can create a new account:** Even though David and Diana are using
312 the same key, David should be able to create a new account that saves his
313 preferences, and switch to it.

314 **Existing multi-user models**

315 This chapter describes the conceptual model, user experience and design ele-
316 ments used in various non-Apertis operating systems' support for multiple users,
317 because it might be useful input for decision-making. Where available, it also
318 provides some details of the implementations of features that seem particularly
319 interesting or relevant.

320 **Switchable profiles without privacy**

321 The simplest multi-user model can be found in platforms such as Windows 98
322 and the Sony PlayStation 3. In these systems, certain settings and other pieces
323 of application data (such as documents and saved games) are stored separately
324 for each user, but there is no privacy or protection between users: each user can
325 easily access other users' accounts.

326 One variant of this is where no authentication is required to access a different
327 account, as on the PlayStation 3: a user selects their name from a list, and
328 there is nothing preventing them from selecting a different user's name instead.
329 Similarly, an unauthorized user can identify themselves as any authorized user
330 and gain access.

331 Another variant of this is where there is meaningful authentication (e.g. a login
332 step with a password), but authenticating as *any* user is sufficient to access *all*
333 users' private files. For instance, Windows 95 offered login authentication, but
334 did not support filesystems with user-level permissions. As a result, unautho-
335 rized users were prevented in principle (in practice, the login step was easily
336 circumvented), but each authorized user had the technical capability to read
337 and write any other user's files by navigating to the appropriate directory.

338 Both variants of this model are simple to implement, and provides straightfor-
339 ward semantics. Their disadvantage is that they do not meet typical privacy
340 expectations for a modern operating system: users can impersonate one another,
341 read each other's private files, and even alter each other's private files. As such,

342 it is only suitable for an environment in which every user of the system fully
343 trusts every other user of the system (and, for the first variant, everyone with
344 physical access to the system).

345 We anticipate that these simple use-cases will be appropriate for some, but not
346 all, Apertis systems: for example, they might be appropriate for a family car
347 where the installed apps do not handle particularly sensitive information. In
348 other Apertis systems, stronger privacy/protection between users is likely to be
349 required.

350 **Typical desktop multi-user**

351 Many modern desktop/laptop operating systems (such as Windows 10, Mac OS
352 X, and various open source desktop environments on Linux and BSD platforms)
353 have a similar model for how multiple users are handled. Apertis shares many
354 software components with the GNOME 3 desktop environment (as used in, for
355 instance, Debian GNU/Linux and Fedora Linux), so we will use GNOME on
356 Linux as our primary example of this type of environment.

357 On Unix-derived systems such as Linux and Mac OS X, each user account is
358 typically represented by one Unix uid, corresponding to their intended use in
359 all Unix systems.

360 **Basic multi-user: log out, log in as another user**

361 The most basic form of multi-user support is considerably older than graphical
362 user interfaces, and is implemented in most current desktop/laptop operating
363 systems. The system boots to a login prompt at which the user can choose their
364 user account (for instance by choosing from a list or by typing its name), and
365 authenticate in some way (typically with a password, but many authentication
366 mechanisms are possible).

367 Each user has their own set of data files and configuration. To provide privacy
368 between user accounts, the system tracks the ownership of user files, and either
369 denies access to other users' files by default, or can be configured to do so.

370 To switch between users, the first user must log out, ending their session; this
371 typically also terminates most or all of their user services. Ending their session
372 presents another login prompt, at which the second user can log in.

373 In a typical implementation on Linux systems with the X11 windowing system,
374 a system service (a “display manager”, such as GNOME's GDM) starts an X
375 display and uses it to show the graphical login prompt. When the first user
376 logs in, their uid is granted access to the X display, which is taken over by their
377 session. At the end of their session, the display manager terminates the X server,
378 and starts a new X server for the next login prompt.

379 Systems which offer this model can easily support the simpler models from
380 [Switchable profiles without privacy] as trivial cases of this model: they can

381 implement the PlayStation 3-like model by omitting the authentication step
382 after choosing a user, or the Windows 95-like model by giving each authorized
383 user access permissions for other users'files.

384 **Fast user switching: switching user without logging out**

385 A refinement of the above model for systems with enough memory is to offer
386 more than one parallel login session, with one active login session and any num-
387 ber of inactive sessions. This is commonly referred to as *fast user switching*.

388 Again, most current desktop/laptop operating systems offer this in some form.
389 The first user chooses a “Switch User...”option from a menu; this optionally locks
390 the first user's session (for instance by locking their screensaver), and switches
391 to a login prompt at which the second user can log in. To switch back, the
392 second user uses “Switch User...”to access another login prompt, at which a
393 third user can log in, and so on. Several users can share the system, with up to
394 one active session and any number of inactive sessions (limited by system RAM,
395 and optionally an arbitrary limit on the number of users).

396 If the user logging in at the login prompt already has a login session, then the
397 system detects that, and instead of starting a new session, it switches back to
398 the existing session, automatically unlocking the screensaver if required. When
399 a user logs out, their session is replaced by a login prompt at which any user
400 can log in.

401 Designers typically treat this model as a superset of the simpler model in **Basic**
402 **multi-user: log out, log in as another user**: in practice, implementations of fast
403 user switching also offer the non-concurrent log-out/log-in arrangement as a
404 trivial case. Similarly, as in **Basic multi-user: log out, log in as another user**,
405 implementations of this model can easily support the models from [Switchable
406 profiles without privacy] as trivial cases.

407 In GNOME's GDM display manager, the first session takes over the X server
408 originally used for the login prompt, the same as in **Basic multi-user: log out,**
409 **log in as another user**, this runs on a Linux virtual console, traditionally tty7.
410 The “Switch User...”option causes the display manager to run a new X server
411 on a different virtual console, typically tty8, and switch to it; the second user's
412 session takes over that X server, and so on, allocating a new virtual console
413 and running a new X server each time. If a user logs out, the display manager
414 remains on the same virtual console, but runs a new X server for the login
415 prompt. If the user logging in at the login prompt already has a login session,
416 instead of taking over that X server for a new session, the display manager
417 switches to the appropriate virtual console for the existing session. The X server
418 with the login prompt remains in the background, and is re-used the next time
419 a login prompt is required, instead of starting a new X server: for example, a
420 system where three users Alice, Bob and Chris repeatedly switch between their
421 accounts would reach a “steady state”with four X servers on four virtual consoles
422 (corresponding to Alice, Bob, Chris, and the login prompt).

423 Once two or more users have logged in, this model provides very rapid switching
424 between them: none of their applications or user services need to be terminated
425 or restarted. It also eliminates any loss of transient “context” such as notifica-
426 tions or window positions, without needing to implement state-saving. However,
427 it uses a significant amount of memory: because inactive users’ applications are
428 not terminated, two alternating users could need up to twice as much memory
429 as a single user. Similarly, because the inactive users’ applications are not termi-
430 nated or paused, merely disconnected from input and display devices, they can
431 continue to consume other resources, such as CPU time and network bandwidth:
432 a misbehaving application in Alice’s session can cause Bob’s session to appear
433 slow.

434 **Multi-user desktops with multi-seat support**

435 Some systems, in particular the systemd-logind component used in Apertis, can
436 be used to extend the model in **Basic multi-user: log out, log in as another**
437 **user** by offering several so-called “seats” as defined in **Multi-seat logind seats**. A
438 logind seat is a collection of display and input devices intended to be used by
439 a single user, offering the equivalent of section **Basic multi-user: log out, log in**
440 **as another user** independently on each logind seat. Similarly, a system can offer
441 “fast user switching” (**Fast user switching: switching user without logging out**
442 on some or all of the available logind seats).

443 GNOME’s GDM display manager switches between virtual consoles on the first
444 logind seat, in exactly the same way as section **Fast user switching: switching**
445 **user without logging out**. On the second and subsequent logind seats, it behaves
446 as described in **Basic multi-user: log out, log in as another user**, with this logind
447 seat’s X server remaining visible regardless of the current virtual console, and
448 does not offer “fast user switching”.

449 **Android 4.2+**

450 Recent versions of Android have gained multi-user support, initially for tablets
451 only, then extended to phones in Android 5.

452 When first started, **Android 4.2**¹ shows a prompt for setting up the first user
453 account. The first user account is special in that it is considered the administra-
454 tor for the device, and can thus create, remove and assign permissions to other
455 users.

456 Android uses separate Unix user account IDs (uids) for separating applications
457 from each other, so any communication or sharing between applications was
458 already mediated by the Linux kernel and other trusted parts of the Android
459 system software. The multi-user design simply allocates a block of uids to
460 each user, one uid per (user, application) pair: for example, the first user (user

¹<http://developer.android.com/about/versions/jelly-bean.html#android-42>

461 number 0) might receive uids u0a123 and u0a45 for two of their apps, and user
462 number 1 might receive uids that include u1a67.

463 Because applications are already isolated from one another by their differing
464 uids, all interaction between apps is mediated by trusted processes, so those
465 trusted processes were adapted to take the user into account when deciding
466 permissions. Similarly, because apps conventionally use Android-specific APIs
467 to access user data, adapting those Android-specific APIs to take the user into
468 account is straightforward: an application making an API call that previously
469 listed *all* online service accounts will now only be told about the appropriate
470 user's online service accounts.

471 Authentication is through the usual means used by Android: each user gets
472 their custom lock screen and, depending on that user's settings, types in a PIN, a
473 password or a pattern connecting dots in a grid for logging in. Icons representing
474 all users are shown in the current user's lock screen, so user switching is a
475 matter of locking the screen (which can be done through the 'quick settings'
476 menu, available in the status bar) and tapping the desired user.

477 From a user interface perspective, this resembles **Fast user switching: switching**
478 **user without logging out** on typical desktop operating systems. However, as
479 an implementation detail, each user's apps are terminated when user switching
480 occurs, so the actual implementation is closer to the "log out / log back in" model
481 (section **Basic multi-user: log out, log in as another user**).

482 Some settings are global to the device, including Wi-Fi networks. All users
483 can change these settings, apparently, and those changes will affect every other
484 user. User settings and data are kept separate from each other's. The list of
485 applications in the user's launcher is separate for each user, but application files
486 are only downloaded the first time a user asks that application to be installed,
487 to save space.

488 Because Android provides custom API for everything the application does, the
489 storage and reading of data and settings for each user is done automatically by
490 their APIs. That means applications did not have to be modified for supporting
491 multi-user: the fact that they already use Android APIs to obtain directory
492 paths and save files ensures that they are saved to the proper place.

493 **Multi-user support in the Tizen 3 automotive platform**

494 The multi-user architecture designed for [Tizen 3] in an automotive environment
495 was presented at FOSDEM 2015.

496 At a conceptual level, Tizen applications can either be installed system-wide or
497 for a particular user. Guest users can only use system-wide applications; it was
498 not clear from the presentation whether only preinstalled applications can be
499 system-wide, or whether separate installable applications can also be installed
500 system-wide. If installed for a particular user, the application's files are copied

501 into that user's home directory, contrasting with the centralized app storage
502 used "behind the scenes" in this design document and in Android.

503 The Tizen model is designed for a "multi-seat" environment as described in **Multi-**
504 **user desktops with multi-seat support**, where several sets of grouped devices (a
505 display, its attached touchscreen input device, and perhaps USB sockets and/or
506 a headphone jack located near that display) are all attached to the same com-
507 puter as peripherals; this is an attractive model if the system is powerful enough
508 to provide acceptable performance on all seats, but comes with higher perfor-
509 mance requirements than some of the potential classes of requirements addressed
510 by this document. In particular, there is a focus on the ability to move concur-
511 rent applications seamlessly from one screen to another, following a user who
512 moves from one seat to another.

513 In the Tizen model, all users share a single compositor, which manages all seats'
514 displays and input devices, resulting in the compositor being required to act
515 as part of the TCB for security between users (see [Trusted components]). As
516 discussed further in **Graphical user interface and input**, we do not recommend
517 this approach while using X11 for GUI services.

518 There is a single privileged user in the Tizen system, and only that user can
519 configure certain shared resources such as wireless networking and Bluetooth.
520 This seems an unnecessarily limiting model for a car that might be shared
521 between two or more primary drivers, for example in a family. It is intended
522 that this user will eventually be able to launch applications on seats that are
523 currently in use by other users.

524 The API model in Tizen appears to involve system services such as the media
525 server and thumbnail generation service not only acting on behalf of users to ful-
526 fill requests, but running as 'root' so that the same application can write directly
527 into multiple users' home directories. We recommend avoiding this practice: it
528 puts all of those services into the TCB for each layer of the security model (secu-
529 rity between users, security between apps and security between system services),
530 greatly increasing the amount of security-sensitive code in the system and the
531 potential impact of a bug or security flaw.

532 The presentation mentioned adding the user ID as an explicit parameter in
533 IPC (inter-process communication) calls from applications to system services
534 so that the system service will act on behalf of the appropriate user. This
535 could be made to work securely by verifying that the actual user ID matches
536 the one in the IPC call, but is a potentially dangerous approach: if a naive
537 implementation trusts the given parameter and does not verify it, a malicious
538 application could easily subvert that implementation. We recommend avoiding
539 "user ID" parameters in APIs: if the service can determine the user ID in a
540 secure way, then the parameter is unnecessary, and if it cannot, this approach
541 brings the calling application into the TCB for security between users (with the
542 practical result that all or nearly all applications would end up in the TCB,
543 greatly increasing the system's attack surface).

544 Approach

545 Because this document does not define precise requirements or use-cases for
546 the system, this section outlines multiple possible approaches to several design
547 questions. The choice between these approaches must be made based on concrete
548 requirements.

549 The principle of least-astonishment

550 One valuable general design principle is that, when a user carries out an action,
551 it should be easy to predict the outcome. In the context of a multi-user system,
552 this implies various more concrete principles, such

- 553 • sharing should not occur when a user would not expect it to; this “over-
554 sharing” is likely to lead to users distrusting the system and being unwilling
555 to store private data in it, even if that would be advantageous
- 556 • sharing should occur when a user would expect it to; if it does not, users
557 will be inconvenienced by having to copy data manually between different
558 contexts
- 559 • performing a similar action in different contexts should have a similar
560 result

561 Levels of protection between users

562 There is a spectrum of possible sets of requirements for privacy and integrity
563 protection between users: a strongly protected model similar to the one detailed
564 in section [Typical desktop multi-user], a model with no protection at all as
565 described in [Switchable profiles without privacy], or anything in between (e.g.
566 with protection between users in general, but certain categories of data explicitly
567 shared).

568 The desired level of protection depends on the user, but we could also decide
569 that Apertis will only support a subset of the possible range, and an OEM could
570 decide that they will only support a subset of the range allowed by Apertis.

571 In use-cases that involve differently-privileged users, the desired level of protec-
572 tion might vary between users within a system: for instance, the main users of
573 a car might opt for a setup in which switching from one main user to another
574 does not require authentication, but switching from a “guest” user to a main user
575 does.

576 For each set of requirements, we aim to minimize the “friction” in switching be-
577 tween users, subject to whatever minimum is imposed by the requirements –
578 stronger privacy and integrity protection comes with a higher minimum “fric-
579 tion”. For example, if users are to be protected from each other, then switching
580 between users must include an authentication step, whereas if there is no ef-

581 fective protection (privilege boundary) between users, switching between users
582 merely requires choosing the desired user account.

583 As a general design principle, design documents for concrete use cases should ad-
584 dress the “strongest”supported protection between users, because that imposes
585 the most difficult privacy/integrity requirements. Secondly, they should con-
586 sider the “weakest”supported protection between users, because that imposes
587 the most general sharing requirements: ideally, this is just a trivial case of the
588 high-privacy version, with some of the “pain points”omitted, but it does intro-
589 duce new requirements for the ability to pass data between users. All other
590 levels of privacy/integrity protection can be represented as somewhere between
591 those extremes.

592 As a compromise plan if we find situations that cannot be solved in a higher-
593 privacy model, it is possible to relax our requirements to declare the highest-
594 privacy use cases to be out of scope.

595 **User accounts: representing users within the system**

596 There are three possible approaches to representing users in a Linux system.

597 **Sharing one uid between all users**

598 In this approach, all user applications and user services run under the same
599 uid. The system defines its own proprietary “user account”concept, and all
600 components that access user-specific data must ensure that they access the
601 correct user’s data, disallowing access to other users’data if appropriate.

602 This has the potential to make transitions between users very easy: the “current
603 user”is simply a variable within each application or service. However, it places
604 a great deal of trust on each of these components, including every third-party
605 (user-installable) application that accesses user-specific data. If the system’s se-
606 curity model is that users can be protected from each other, then in effect, all
607 of these components are included in the trusted computing base; if the require-
608 ments do not include protection between users, then distinguishing between
609 users is not required for security, but is still required for correctness. In prac-
610 tice, we anticipate that not every component would discriminate between users
611 correctly.

612 This approach also has practical problems for the re-use of existing open source
613 components, which assume the traditional use of one uid per user. Having to
614 modify all of these components, with a complex change that is unlikely to be ac-
615 cepted by their upstream developers, would significantly reduce the competitive
616 advantage derived from their use.

617 As a result of these disadvantages, we do not recommend this approach for
618 Apertis. It would only be viable if all of the following are true:

- users are not protected from each other, and this will not change in future development
- user-specific data is minimal, only needs to be accessed via Apertis-specific APIs, and this will not change in future development
- it is not considered to be a significant problem if third-party applications and services do not consistently distinguish between users, and this will not change in future development

An additional consideration for this approach is that it potentially alters a large number of interfaces (such as D-Bus method calls) to have a parameter for the user account to be affected. If changing requirements result in switching to the “one uid per user” or “many uids per user” models in future, such that the correct user account is implicit in the uid, then this vestigial parameter will remain in the interface, making the interface more complex than is required.

If the form of the additional parameter resembles the numeric or string form of a uid, then this could even lead to security issues, for instance if a component trusts the explicit user-account parameter and ignores the actual uid.

If this approach is taken, then we recommend reducing the confusion caused by naming the additional parameter something more similar to “profile” than “user”. If the system is later extended to have one uid per user, rendering the parameter vestigial, we recommend giving it a neutral, constant value that does not match any user account name, such as “default”.

One uid per user

The traditional Unix design which motivated the uid concept is that each user account is represented by one numeric uid.

Because each process (i.e. each application or service) starts with a particular uid, and processes without administrative privileges cannot change their uid while running, this approach requires that user-switching involves starting new processes for the new user.

The major advantage of this approach is that it is how the existing components in the system, including the Linux kernel, are designed to operate. In particular, the Linux kernel provides privacy and integrity protection between uids.

We recommend this approach for Apertis.

Multiple uids per user

Android uses a design involving multiple uids per user, one per app or set of related apps, as described in [Android 4.2+](#). This allows the Linux kernel’s privacy and integrity features to be used to protect apps from other apps, even within a user session. However, in Apertis, this advantage is redundant, since

656 we already use a different kernel feature (AppArmor) to provide privacy and
657 integrity protection between apps.

658 The major disadvantage of this approach is that it requires every interaction
659 between dissimilar apps to be mediated by a system-level component. Within
660 the context of Android, this is not a problem, since Android applications and
661 services are expected to use Android-specific APIs in any case. However, Apertis
662 re-uses existing open source components where appropriate; these components
663 would have to be modified to cope with crossing privilege boundaries when they
664 communicate with different uids, which, as in the “one shared uid” approach,
665 would reduce the value of re-using these components.

666 We do not recommend this approach for Apertis.

667 **Creating and managing user accounts**

668 Based on the description of desired use case scenarios, Apertis understands the
669 main means of identifying and authenticating a user will be through a external
670 medium, such as a car key.

671 If runtime user-switching is required, a secondary form of authentication is likely
672 to be required. This could be done via a password (or equivalent, such as a PIN
673 or touchscreen swipe pattern), via biometrics such as fingerprint, face or voice
674 recognition, or by verifying possession of a near-field communication device such
675 as a mobile phone.

676 As previously noted, depending on manufacturer and consumer requirements,
677 there is the possibility of simpler authentication schemes for less privacy-
678 conscious users; for instance, a manufacturer or consumer could choose to
679 relax the security model to one where the external medium is sufficient to
680 authenticate as any registered user selected from a menu.

681 A registration process will be required, to associate authentication tokens with
682 user accounts: one way this could work is detailed in this section.

683 **Registering the users**

684 When the system in run for first time each user in turn will follow the following
685 procedure:

- 686 1. A user is authenticated by an external medium, such as a car key
- 687 2. The Apertis system starts up and recognizes that it is a new unregistered
688 user
- 689 3. A wizard is displayed to register the new user
- 690 4. The user enters whatever information is needed to set up their user ac-
691 count, such as their name

- 692 5. The user is given the option of registering a password or other authentication tokens to be used for keyless authentication (for user switching,
693 mainly)
694
- 695 6. Alternatively the wizard can continue from here on to register email and
696 web accounts the user may be interested in
- 697 In case there are more users than keys available, new keys will need to be
698 acquired.

699 **The first user to be registered is special**

700 It's important that at least one user be able to perform administrative tasks,
701 such as wiping out all of the data, removing users, and so on. One practical
702 solution to this is that the first user to be registered is considered special and
703 be able to perform these tasks and is also able to give these privileges to other
704 users as they see fit, so that more users would be able to perform administrative
705 tasks.

706 One analogy used in the security literature is that the system “imprints” on the
707 first user seen, in the same way that a duckling imprints on its parent. A
708 refinement of this model is that deleting all users resets the system to a state
709 in which the next user created will be privileged, the so-called “[Resurrecting
710 duckling²](https://www.cl.cam.ac.uk/~fms27/duckling/)” model.

711 Frank Stajano and Ross Anderson. *The Resurrecting Duckling: Security
712 Issues for Ad-hoc Wireless Networks*. In B. Christianson, B.
713 Crispo and M. Roe (Eds.). *Security Protocols, 7th International
714 Workshop Proceedings*, Lecture Notes in Computer Science, 1999.

715 **Graphical user interface and input**

716 This section explores several potential models for managing input and output.

717 The basic infrastructure component for Wayland is a *compositor*, which is responsible for mapping application-supplied surfaces (windows) into the visible display, routing input events to those surfaces, and applying any visual effect with a larger scope than an individual application, such as animated transitions between applications.

722 The Wayland compositor is part of the TCB for security between apps: it is responsible for imposing a boundary between the apps that communicate with it, and preventing them from carrying out undesired actions such as reading each other's input or taking screenshots of each other's windows. Depending on the design and implementation, it may also need to be part of the TCB for security between users.

²<https://www.cl.cam.ac.uk/~fms27/duckling/>

728 In the current design, the compositor being used is agl-compositor. It is a
729 Wayland compositor based on `libweston`.

730 **Single compositor**

731 One possible model is to have a single compositor which starts on boot, runs un-
732 til shutdown, and is directly responsible for compositing all application surfaces.
733 This model would be appropriate if there is only one uid shared by all users as
734 described in section **Sharing one uid between all users**, since in that model there
735 is no OS-level isolation between user accounts in any case. It could potentially
736 also be used in a design where each user has their own uid, by running the
737 compositor with a non-user-specific uid.

738 The major disadvantage of this situation is that it places the user-level composi-
739 tor into a trusted position: it would become part of the trusted computing base
740 for separation between users (see [Trusted components]).

741 As a general design principle, the less code is in the trusted computing base (for
742 any given layer of security),

743 the better; this conflicts with the user-level compositor's broad role in medi-
744 ating between apps, including animated transitions, copy/paste functionality,
745 on-screen keyboard handling and so on.

746 **Switching between compositors**

747 The traditional design for user-switching in X, as described in **Basic multi-user:**
748 **log out, log in as another user** and **Fast user switching: switching user without**
749 **logging out**, is to start a new X server for each user session and switch between
750 them, for instance by using the Linux kernel's "virtual console" facility, or by
751 dynamically attaching/detaching the X servers to the video device.

752 A possible implementation for this scenario is to run multiple session compos-
753 itors, switching access to the video output between them, and not having a
754 system compositor.

755 In this model, the transition between users would involve `systemd-logind` revok-
756 ing the old session compositor's control over the display ("DRM master" status)
757 and over input devices, and giving control to the new session compositor. This
758 could be done at any point in the transition: before, after or during an animated
759 transition.

760 The major disadvantage of this design is that switching between virtual consoles
761 is an all-or-nothing operation: the system either displays a frame from one
762 compositor or a frame from another, but it cannot combine two (for instance
763 by overlaying them, with transparent regions). It is also not instantaneous, and
764 would have to be disguised by having a transition where several consecutive
765 frames are allowed to be the same.

766 For some UX designs, this would not matter. For example, if a designer specifies
767 that the first user’s session should “fade out” to a black screen or some sort of
768 “please wait…” placeholder, or move off-screen, then the system could switch to
769 a matching frame in the new compositor, wait for the switch to occur, and have
770 the second user’s session “fade in” or move in from off-screen. Similarly, if the
771 UX for user-switching involves a menu from which the new user is chosen, then
772 that menu could be used as a fixed point around which to anchor the transition.

773 However, if the desired transition has the two users’ sessions overlap –for instance,
774 a full-screen cross-fade from one to the other, or any animated movement that
775 has both sessions exist on-screen at the same time –then it would be difficult to
776 achieve these effects in this design without essentially copying a static screen-
777 capture of one session into the other session. Similarly, if the desired transition
778 has smooth movement from beginning to end –for example, smooth horizontal
779 scrolling with the conceptual model that the other user’s session is “just off-
780 screen” –then the only practical points at which to do the virtual console switch
781 would be at the very beginning or at the very end; either way, this would likely
782 result in a few frames of non-responsiveness at a time when the user might
783 reasonably expect the system to be responsive.

784 Copying a screen-capture of one session into the other session is also a potential
785 privacy risk, since it results in the screen contents crossing the trust boundary:
786 it would be technically possible for the second user’s session to save the captured
787 image.

788 **Switching between compositors with a system compositor**

789 Because Wayland does not require clearing the framebuffer during switching,
790 another possible approach would be to use a system-level compositor without
791 nesting, used for transitions, and optionally for startup and shutdown. At any
792 given moment, either the system-level compositor or a session compositor would
793 be active (have control over input and output), but never both.

794 In this model, as in [Switching between compositors], the transition between
795 users would involve systemd-logind revoking the old session compositor’s control
796 over the display (“DRM master” status) and over input devices; however, instead
797 of immediately giving control to the other session, instead it would give control
798 to a special-purpose system-level compositor which would perform the transition,
799 and then in turn hand over to the new session. This system-level compositor
800 could capture the current screen contents as a starting point for the animated
801 transition, if desired; as in [Switching between compositors], the screen contents
802 would cross a privilege boundary, but unlike [Switching between compositors],
803 the other side of the privilege boundary in this design is a trusted process.

804 The new session compositor could be started without direct access to the display
805 (it would not yet be the “DRM master”), and instructed to draw its initial state
806 into a buffer; recent Linux kernel enhancements mean that it could use in-GPU
807 processing and memory for this drawing operation, without having control over

808 what is displayed. The system-level compositor would use that buffer as the
809 endpoint of its animated transition. On completing the transition, it would
810 instruct systemd-logind to grant full display and input access to the new session
811 compositor.

812 As a result of its role in user-switching, the system-level compositor used for
813 the transition would potentially be part of the TCB for security between users.
814 However, its functionality would be minimal: because it would not be active
815 during normal use, only during transitions, it would not necessarily need to
816 process input at all, and its output handling would be limited to performing the
817 animation from the old to the new screen contents.

818 **Switching between users**

819 If runtime switching between users is required, there is a spectrum of possible
820 approaches.

821 At one extreme is the simplest form of the approach described in section 4.2.1,
822 where we terminate all of the newly inactive user’s apps and user services (any-
823 thing that is user-specific), and only non-user-specific processes (system services)
824 continue to run. That has the lowest possible memory and CPU overhead: there
825 is going to be a small amount of overhead during the necessary “grace period”
826 while we let the inactive user’s apps save their state before killing them, but this
827 is minimized.

828 At the opposite extreme is the “fast user switching” as described in section 4.2.2,
829 in which the inactive user’s entire session, including GUI apps, user services,
830 games, and infrastructure components such as the window manager and session
831 compositor continue to run, with the only difference being that they are discon-
832 nected from the input and display hardware. That has considerable overhead:
833 in the worst case, where we assume that system services are negligible when
834 compared with per-user components, switching between two users could double
835 the memory and CPU consumption.

836 We can choose various points along that spectrum depending on OEM and
837 customer requirements. If we can terminate all of the inactive user’s apps and
838 the majority of their user services, the result is close to the first extreme –for
839 example, this could be based on an “agents continue to run across user-switching”
840 flag in the app manifest, perhaps implemented as an Android-style “permission”
841 . App-store curators could carry out more thorough validation on services that
842 request that flag, to ensure that they will not have an adverse performance
843 impact.

844 If we can terminate all of their apps but must leave *all* of their user services
845 running, we get closer to the second extreme. The closer we are to the second
846 extreme, the higher our hardware requirements for a given performance level
847 will be.

848 If we terminate at least some of the newly inactive user’s processes, a second

axis of variation is how much overlap we are prepared to tolerate between the sessions: to allow those processes to save their current state, a “grace period” will be required between notifying those processes that they must exit, and actually terminating them.

One approach is to disallow overlap entirely, and not start the transition until the inactive user’s session has completely ended, with a “please wait…” message while their processes shut down. However, this maximizes latency and user-visible disruption. To reduce the time required to switch between users, it might be desirable for these processes to continue to run concurrently for a short time, in parallel with starting the newly active user’s session. There is a trade-off here: the more CPU time is consumed by the newly inactive user’s processes, the less is available to display a smooth animated transition to the newly active user and launch *their* processes. This could be mitigated by de-prioritizing the CPU and bandwidth consumption of the inactive user’s apps, at the cost of extending the necessary “grace period” for a given amount of state-saving activity: for example, if an app’s state-saving procedure would normally take 50% of the CPU for 0.1 seconds, throttling that app to 5% of the CPU would make its shutdown take 1 second.

Preserving “core” functionality across user-switching

If user-switching during use is supported, then certain features of the system must continue to work during and after the user switching operation.

For example, navigation-related notifications (notifying the driver that they should turn off their current route soon, that the speed limit will change soon, etc.) are time-sensitive, and it would be reasonable to require that these notifications are not interrupted or delayed, even if user switching takes place just before or even during the notification.

Further examples of background features that might be in the category that must not be interrupted include media playback (if the driver is listening to music, it would be reasonable to require that playback is not stopped or disrupted by user switching, although interrupting “now playing…” notifications might still be acceptable) and incoming phone or VoIP calls.

These features cannot be assumed to be a fixed part of the operating system: for example, it should be possible to have uninterrupted media playback via a third-party audio streaming app, such as one for last.fm or Spotify, or uninterrupted VoIP call notifications for a third-party VoIP implementation.

Conversely, essential operating system features such as preinstalled or non-removable apps are not necessarily all in the category of features that must continue to work during user-switching: for example, incoming email notifications are less time-critical than calls, and it is likely to be acceptable for them to be paused during user-switching.

There are several possible approaches to keeping these features working across

a user-switch. Depending on the concrete requirements and use cases, we could choose one of these approaches for the whole system, or choose some combination of them for different apps and services.

As mentioned briefly above, there is the potential for a subtle distinction between components where an interruption to notifications is unacceptable (for instance, navigation or incoming calls might be in this category), and components where an interruption to functionality is unacceptable, but an interruption to notifications is allowed (or even desirable).

For a possible example of the second category, consider music playback, on a system where a visual notification is triggered when the current track changes. Suppose we switch the current user from Alice to Bob at 12:00:00, at which time track 1 is 2 seconds from ending, and the animated transition takes 4 seconds. It seems reasonable to expect that track 1 must continue to play until 12:00:02, and it also seems reasonable to expect that track 2 must start at 12:00:02 and continue to play smoothly. However, it is not necessarily a requirement that the “now playing track 2” notification cannot be delayed until Bob’s session becomes fully available at 12:00:04; indeed, this might be considered more desirable than having it interrupt the animated transition.

System services

System services (as defined by [System services]) continue to run regardless of what is happening in user sessions, so one possible approach is to put “core” functionality in system services. These could be anywhere from highly privileged to entirely unprivileged; the distinction here is only that they are independent of user accounts.

For example, network management services such as ConnMan are highly-privileged system services, whereas the Avahi name-resolution and service discovery service is system-wide but unprivileged.

If this approach is to be used for third-party installable applications, then we will need to ensure that third-party application bundles can provide system services, in a way that does not allow those third-party application bundles to compromise the overall security of the system.

For components that deal with user-specific data, making the component into a system service requires that the component is trusted to provide the correct privilege separation: for example, if the component has access to multiple users’ private data, it should not reveal one user’s private data to another user unless the system’s security model allows this to happen.

As a general design principle to avoid circular dependencies and unnecessarily tightly-coupled components, lower layers should not rely on higher layers. System services are at a low layer in the stack, so they should not initiate communication with user services or users’ graphical sessions. One common approach to this is to have a component inside each user session whose role is to provide

931 the user interface for a “headless” system service, separating backend logic and
932 system-level configuration (the system service) from user interface presentation
933 and per-user configuration (the user part).

934 **User services continuing to run**

935 User services (as defined by [User services] are inherently per-user. If the end
936 of a user’s login session terminates their GUI applications but leaves some or
937 all of their user services running, this could increase system load (as noted
938 in section [Switching between users]), but would make user services a suitable
939 implementation for features that must run uninterrupted. This could apply
940 either in general, or with restrictions (for example, some subset of the inactive
941 user’s user-services could continue to run, perhaps according to a “flag” in their
942 associated app manifests).

943 **Distinguishing between the driver and other users**

944 Because the driver is the primary user of the system, one possible refinement
945 of this requirement would be to say that core functionality associated with the
946 driver cannot be interrupted, and must retain its ability to display notifications,
947 but that switching may interrupt functionality associated with other users. This
948 would limit the additional system load from multiple users: the maximum set
949 of processes running at a given time would be one non-driver’s full session, plus
950 whatever subset of the driver’s processes are considered to be necessary.

951 **Agents**

952 The Apertis design has the concept of “agents”, which are lightweight back-
953 ground processes running on behalf of a user. Depending on the precise re-
954 quirements for agents, they could be implemented as system services, or as user
955 services, or divided between those two categories.

956 **Returning to previous state**

957 Saving and restoring the state of the session is a hard problem in general. Some
958 platforms, such as Android, made it a central piece of their application life cycle
959 management and built it right into the application support for the platform. The
960 fact that Android and iOS have custom platform layers allows them to make
961 this viable.

962 Apertis is not aware of any deployment of OS-level freezing and thawing of
963 processes at the moment, but such a strategy could be investigated in the future
964 for usage in Apertis. For now, having the application itself care about saving
965 and restoring state, even if supported by some high level API, seems to be
966 the more realistic approach. More discussion about this can be found in the
967 [Applications design](#)³ document.

³<https://apertis-website-0b3586.pages.apertis.org/concepts/archive/application/applicat>

968 Application ownership and installation

969 In current app-store platforms such as Apple, Google Play, Steam or PlayStation
970 Store, if you buy an app, it is associated with your personal account (Apple,
971 Google, etc.) and can be downloaded to any device associated with that account,
972 subject to some limits. This is one possible approach to how apps are deployed
973 on Apertis.

974 To avoid wasting space with duplicate application installations, current app-
975 store implementations with multi-user support, such as Android, have chosen
976 to install applications system-wide. If Apertis apps are, conceptually, installed
977 per-user, then we recommend implementing this by keeping a list of apps per
978 user, and merely hiding apps from users who have not “installed” that app. If
979 the user acquires an app that another user has already installed, the system
980 could behave as though it was freshly downloaded, but in fact just stop hiding
981 the system-wide app from the current user: from the user’s perspective, this is
982 indistinguishable from a very fast download and installation.

983 Another potential conceptual model is to treat apps as more like car accessories.
984 You could, for instance, buy a car with metallic paint, or add alloy wheels
985 later; when you sell the car, the feature goes with it. Applying this model to
986 applications, it could be possible to buy a car with the social media app bundle
987 preinstalled, or add the media streaming bundle later, and have the apps go
988 with the car when it is sold. In some respects, this is the more natural model
989 from the implementation point of view: we do not recommend duplicating the
990 app’s executable code and resources, regardless of whether it is conceptually
991 installed per-user.

992 Whichever of these approaches is taken, choosing whether ownership/licensing
993 of the app follows the car or the purchaser is primarily a matter for the app
994 store implementation, not the multi-user design.

995 Summary of recommendations

996 As discussed in [User accounts representing users within the system], Apertis
997 recommends representing each user account as a Unix user ID (uid). The first
998 user to be registered in a new system must be able to perform administration
999 tasks such as system updates, application installation, creation of new users
1000 and setting up permissions –that is discussed in [Creating and managing user](#)
1001 [accounts](#).

1002 There is a range of possible approaches to switching between users, discussed in
1003 section [Switching between users]. This document does not recommend a partic-
1004 ular choice from that range, since it depends on the available hardware resources
1005 and the system’s use-cases and requirements. For budget-limited designs with
1006 significant hardware limitations, we should consider terminating most user-level

ions/

1007 processes while switching to reduce concurrency, or if this is not acceptable, opt
1008 to leave user-switching unsupported; for premium models with more capable
1009 hardware, the more resource-expensive “fast user switching” approach can be
1010 considered.

1011 In **Preserving “core” functionality across user-switching** we outline various possi-
1012 ble approaches to ensuring that “core functionality” is not interrupted by a user
1013 switch. Services that need to stay running after a user switch should have their
1014 background functionality split from their UIs; they can either run as a different
1015 Unix user account ID – a “system service” – or be a specially flagged “user service”
1016 that is not terminated with the rest of the session.

1017 In **Returning to previous state**, Apertis recommends that applications should be
1018 handling saving and restoring of their state themselves, potentially supported
1019 by helper SDK APIs, which means only applications written with Apertis in
1020 mind would work. That recommendation comes from the fact that there is no
1021 solution that would work for all applications.

1022 Ways of having a smooth visual transition when switching users are discussed
1023 in section **Graphical user interface and input**. This should be implemented
1024 according to OEMs needs.

1025 **Switchable profiles without privacy**) [Switching between compositors]:
1026 #switching-between-compositors [Switching between compositors with a sys-
1027 tem compositor]: #switching-between-compositors-with-a-system-compositor
1028 [Switching between users]: #switching-between-users [System services]:
1029 #system-services [Tizen 3]: https://fosdem.org/2015/schedule/event/embedded_multiuser/
1030 [Trusted components]: #trusted-components [Typical desktop
1031 multi-user]: #typical-desktop-multi-user [User accounts representing users
1032 within the system]: #user-accounts-representing-users-within-the-system [User
1033 services]: #user-services