

GDBus Engineering Guide

Architecture, Internals, and Practical Implementation

A comprehensive handbook covering the GLib/GIO D-Bus subsystem: GVariant type mechanics, client proxies, signal dispatching, low-level server vtables, automatic code generation with gdbus-codegen, async event loops, credential enforcement, and debugging tools.

Compiled by: HARDIK

LinkedIn: www.linkedin.com/in/hardikhardik

Written from hands-on systems programming notes, verified across GLib/GIO production deployments, and enriched with internal architectural diagrams.

Table of Contents

Ch.	Chapter & Topic Overview	Page
1	What is GDBus? (Origins, dbus-glib vs. GDBus, libgio Hierarchy, GObject Ref Counting)	2
2	The GVariant Type System in Depth (Type Codes, Format Strings, Tuple Rules, &s vs s)	3
3	Containers: Builders & Iterators (Building Arrays, a{sv} Dicts, iter_loop vs iter_next)	4
4	Connecting & Error Handling (g_bus_get_sync, SASL Auth Lifecycle, GError Discipline)	5
5	Client-Side: Method Calls & Properties (GDBusProxy Internals, Property Caching, Call Lifecycle)	6
6	Client-Side: Signal Subscriptions (g_dbus_connection_signal_subscribe, PropertiesChanged)	7
7	Server-Side: Low-Level VTable Implementation (XML Introspection, Invocations, Property Handlers)	8
8	Server-Side: Ownership & Signal Emission (Name Acquisition, Broadcast Signals, Manual Property Updates)	9
9	The Code Generation Way (gdbus-codegen) (Type-Safe Proxies, Skeletons, Auto PropertiesChanged)	10
10	GMainLoop, Async Operations & Deferred Replies (Async Calls, Slow Work Deferral, Sockets)	11
11	Credential Checking & Security Verification (Sender UID Validation, Polkit, End-to-End Traces)	12
12	Memory Rules, Comparison & Quick Reference (6 Golden Rules, libdbus vs sd-bus vs GDBus, Mental Model)	13

REFERENCE CONVENTIONS

Throughout this guide: **NOTE** highlights critical implementation contracts; **TIP** marks idiomatic GLib shortcuts; **WATCH OUT** warns against subtle memory leaks or crashes; and **EXTRA** provides deep kernel/GLib execution details.

1 What is GDBus?

1.1 Origins and Evolution

GDBus is the official GLib/GIO implementation of D-Bus. It is *not* a wrapper around `libdbus`. It is a completely independent, ground-up reimplementation developed by David Zeuthen and the GLib engineering team starting in 2008, officially released with GLib 2.26 in 2010.

Historically, GTK and GNOME applications used `dbus-glib`, an early binding that wrapped `libdbus`. That legacy binding suffered from fragile multi-threading, lacked support for structured data, and is now entirely obsolete.

Feature Aspect	Legacy <code>dbus-glib</code> (Deprecated)	Modern GDBus (GLib 2.26+ Standard)
Underlying Engine	Thin wrapper around <code>libdbus-1</code>	Independent, native socket implementation
Proxy Object	<code>DBusGProxy</code>	<code>GDBusProxy</code> (with automatic caching)
Method Invocation	<code>dbus_g_proxy_call()</code>	<code>g_dbus_proxy_call_sync()</code> / <code>g_dbus_proxy_call()</code>
Initialization	Mandatory <code>g_type_init()</code>	Not required (deprecated in GLib 2.36+)
Threading Model	Fragile; error-prone locking	Thread-safe; integrated into <code>GMainContext</code>
Type System	<code>GValue</code> / custom demarshallsers	Native GVariant type system
Code Generation	<code>dbus-binding-tool</code> (limited)	<code>gdbus-codegen</code> (generates typed proxies & skeletons)

1.2 Where GDBus Lives in the System Architecture

GDBus is compiled directly into `libgio-2.0.so`. Applications access it via the single umbrella header `<gio/gio.h>` and link using `pkg-config --cflags --libs gio-2.0`.

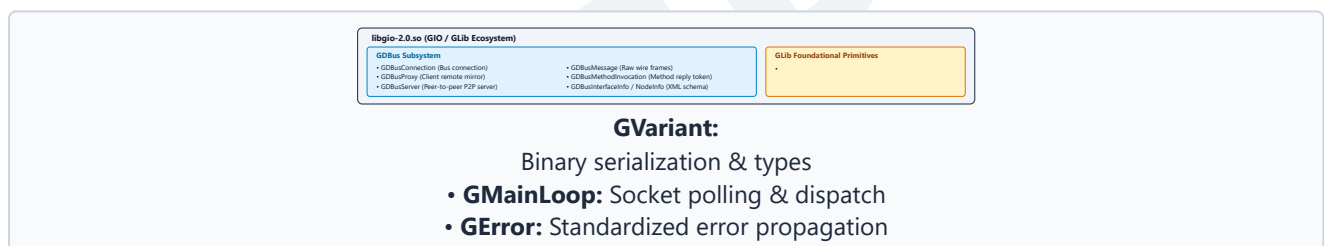


Figure 1.1: The modular components of GDBus and its reliance on foundational GLib runtime facilities.

1.3 GObject and Reference Counting Lifecycle

Almost every primary GDBus entity is an instance of `GObject`. Constructors like `g_bus_get_sync()` and `g_dbus_proxy_new_sync()` hand out references with an initial count of 1. You must increment references with `g_object_ref()` when taking shared ownership and balance every increment with `g_object_unref()`.

NOTE

While `GDBusConnection` and `GDBusProxy` are standard `GObject` types, **GVariant** is a lightweight value-type structure that uses its own independent atomic reference counting functions: `g_variant_ref()`, `g_variant_unref()`, and `g_variant_ref_sink()`.

2 The GVariant Type System in Depth

2.1 GVariant Fundamentals & Constructors

GVariant is the serialization engine for D-Bus messages. It guarantees strong type safety, zero-copy wire parsing, and deterministic memory alignment. Values are created using explicit constructors or concise format strings:

```
// Basic explicit constructors
GVariant *v_i32 = g_variant_new_int32(42);
GVariant *v_u32 = g_variant_new_uint32(100);
GVariant *v_str = g_variant_new_string("hello world");
GVariant *v_obj = g_variant_new_object_path("/org/example/Obj");

// Format strings: Tuples must always use parentheses ( )
GVariant *v_tup1 = g_variant_new("(ii)", 5, 7);
GVariant *v_tup2 = g_variant_new("(si)", "x", 10);
```

2.2 Complete Format String Reference

Char	D-Bus Type	C Data Type	Extraction / Construction Pattern
b	BOOLEAN	gboolean	0 or 1 stored in 32-bit container
y	BYTE	guchar	Unsigned 8-bit byte
n / q	INT16 / UINT16	gint16 / guint16	16-bit signed / unsigned integer
i / u	INT32 / UINT32	gint32 / guint32	Standard 32-bit integers
x / t	INT64 / UINT64	gint64 / guint64	64-bit timestamps and offsets
d	DOUBLE	gdouble	IEEE 754 64-bit floating point
s	STRING	gchar* (Allocated)	Allocates new string; must call g_free()
&s	STRING	const gchar* (Borrowed)	Points directly into GVariant; do not free
o / &o	OBJECT PATH	gchar* / const gchar*	Valid D-Bus object path (allocated vs borrowed)
g	SIGNATURE	const gchar*	Valid D-Bus signature string
h	UNIX_FD	gint32	File descriptor passed via socket ancillary data
v	VARIANT	GVariant*	Boxed polymorphic value (must g_variant_unref)
()	STRUCT / TUPLE	Nested types	Fixed collection of heterogeneous elements
a...	ARRAY	Container	Requires GVariantIter or GVariantBuilder
@...	ANY TYPE	GVariant*	Extracts raw sub-variant without unwrapping

2.3 The Critical s vs &s Distinction

Understanding string ownership prevents crashes and memory corruption:

```
// Pattern A: Allocated copy (s)
gchar *copy = NULL;
g_variant_get(v, "s", &copy);
g_print("Allocated: %s\n", copy);
g_free(copy);

// Pattern B: Borrowed pointer (&s)
const gchar *borrowed = NULL;
g_variant_get(v, "&s", &borrowed);
g_print("Borrowed: %s\n", borrowed);
// DO NOT call g_free(borrowed); valid only while v is alive
```

WATCH OUT: METHOD CALL TUPLE REQUIREMENT

The top-level body of every D-Bus method call or return is strictly a **STRUCT (tuple)**. When invoking a method via `g_dbus_proxy_call_sync()`, you must format parameters as `"(ii)"`. Passing `"ii"` creates two uncontained integers and will fail at runtime.

3 Containers: Builders & Iterators

3.1 Assembling Complex Types with GVariantBuilder

When creating variable-sized collections (arrays, dictionaries), use GVariantBuilder. A builder operates as a three-state machine: initialize with a container type, append entries, and finalize:

```
// 1. Constructing an array of strings ("as")
GVariantBuilder str_builder;
g_variant_builder_init(&str_builder, G_VARIANT_TYPE("as"));
g_variant_builder_add(&str_builder, "s", "eth0");
g_variant_builder_add(&str_builder, "s", "wlan0");
GVariant *array_v = g_variant_builder_end(&str_builder);

// 2. Constructing an a{sv} dictionary (D-Bus Property hashmap)
GVariantBuilder dict;
g_variant_builder_init(&dict, G_VARIANT_TYPE("a{sv}"));
g_variant_builder_add(&dict, "{sv}", "Volume", g_variant_new_uint32(80));
g_variant_builder_add(&dict, "{sv}", "SpeakerName", g_variant_new_string("Front-Left"));
g_variant_builder_add(&dict, "{sv}", "Muted", g_variant_new_boolean(FALSE));
GVariant *props_dict = g_variant_builder_end(&dict);
```

3.2 Traversing Containers with GVariantIter

To deconstruct arrays or maps, initialize a GVariantIter over the container:

```
// Iterating an a{sv} dictionary with borrowed keys
GVariantIter iter;
g_variant_iter_init(&iter, props_dict);
const gchar *key = NULL;
GVariant *value = NULL;

while (g_variant_iter_loop(&iter, "{&sv}", &key, &value)) {
    gchar *type_str = g_variant_print(value, TRUE);
    g_print("Key: %-12s | Value: %s\n", key, type_str);
    g_free(type_str);
}
```

3.3 g_variant_iter_loop() vs. g_variant_iter_next()

g_variant_iter_loop()

- Automatically unrefs previous values each cycle.
- Frees final value when loop completes naturally.
- **Never manually unref items.**
- Cannot use early break (causes memory leaks).

g_variant_iter_next()

- Allocates fresh references for each item.
- **You must explicitly free/unref each element.**
- Safely supports early loop termination and break statements.

WATCH OUT: EARLY BREAK LEAKS IN ITER_LOOP

If you break out of a `g_variant_iter_loop()` before it returns FALSE, the internally cached GVariant from the active iteration is leaked, and remaining elements are never cleaned up. If you need early exits, use `g_variant_iter_next()` instead.

TIP: SUB-TREE OBJECT EXTRACTION

When iterating over (ao) (a tuple containing an array of object paths), use `GVariantIter *iter; g_variant_get(res, "(ao)", &iter);`. Remember to free heap-allocated iterators via `g_variant_iter_free(iter)` when finished.

4 Connecting & Error Handling

4.1 Establishing a Bus Connection

Applications connect synchronously using `g_bus_get_sync()`, passing either `G_BUS_TYPE_SYSTEM` or `G_BUS_TYPE_SESSION`:

```
GError *error = NULL;
GDBusConnection *conn = g_bus_get_sync(G_BUS_TYPE_SYSTEM, NULL, &error);

if (!conn) {
    g_printerr("Connection failed: %s\n", error->message);
    g_error_free(error);
    return 1;
}

const gchar *unique_name = g_dbus_connection_get_unique_name(conn);
g_print("Connected! Unique name: %s\n", unique_name);

// Free connection when shutting down
g_object_unref(conn);
```

4.2 Internal Connection & SASL Handshake Sequence

When `g_bus_get_sync()` is called, GDBus executes a complete connection lifecycle behind the scenes:

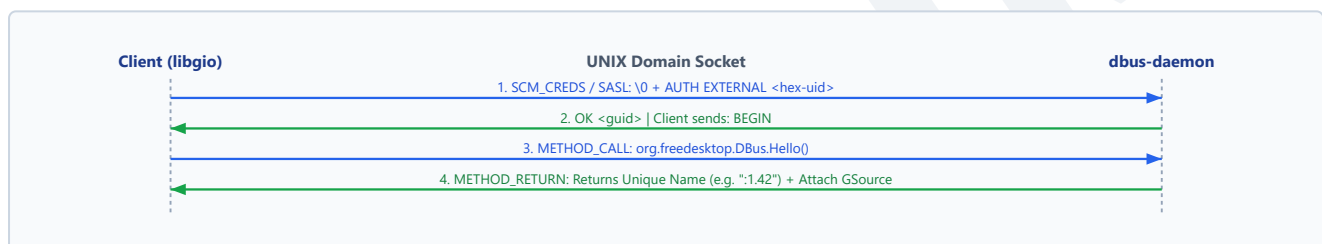


Figure 4.1: The multi-step low-level authentication and initialization flow of `g_bus_get_sync()`.

4.3 Strict GError Discipline

GLib uses the `GError` structure (containing error domain quark, numeric code, and message). Follow these strict discipline rules to avoid application aborts:

WATCH OUT: PASSING DIRTY GERROR POINTERS

Always initialize error pointers to NULL: `GError *error = NULL;`. Never pass an already-set `GError*` to another GLib function without resetting it first. GLib will trigger a hard assertion crash if you pass a pointer pointing to existing memory. Use `g_clear_error(&error)` to cleanly free and reset.

TIP: ASYNC CONNECTION ALTERNATIVE

For UI applications where blocking the main thread during socket connection is prohibited, use `g_bus_get(G_BUS_TYPE_SESSION, cancellable, on_bus_connected, user_data)`.

5 Client-Side: Method Calls & Properties

5.1 Instantiating a GDBusProxy

GDBusProxy mirrors a remote service object. Upon instantiation, it automatically invokes `Introspect()` to inspect interfaces and `Properties.GetAll()` to prime its local property cache:

```
GError *error = NULL;
GDBusProxy *proxy = g_dbus_proxy_new_sync(
    conn,
    G_DBUS_PROXY_FLAGS_NONE,      /* flags */
    NULL,                        /* interface info (auto) */
    "com.example.AdderService",  /* well-known bus name */
    "/com/example/Adder",        /* object path */
    "com.example.Adder",         /* interface name */
    NULL,                        /* cancellable */
    &error
);
```

5.2 Invoking Remote Methods via Proxy

Calling `g_dbus_proxy_call_sync()` serializes parameters into a `METHOD_CALL` frame, transmits it over the socket, and blocks the calling thread while pumping the inner context until a reply or timeout occurs:

```
// Invoke Add(5, 7) -> Returns reply tuple "(i)"
GVariant *params = g_variant_new("(ii)", 5, 7);
GVariant *result = g_dbus_proxy_call_sync(
    proxy,
    "Add",
    params,
    G_DBUS_CALL_FLAGS_NONE,
    -1,
    NULL,
    &error
);

if (!result) {
    g_printerr("Call failed: %s\n", error->message);
    g_error_free(error);
    return 1;
}

gint32 sum = 0;
g_variant_get(result, "(i)", &sum);
g_print("Remote computation result: %d\n", sum);
g_variant_unref(result);
```

5.3 Reading Cached Properties

Because the proxy maintains an internal hash table of properties, reading cached attributes requires zero round-trip bus latency:

```
GVariant *prop = g_dbus_proxy_get_cached_property(proxy, "Version");

if (prop) {
    const gchar *ver = NULL;
    g_variant_get(prop, "&s", &ver);
    g_print("Daemon Version: %s\n", ver);
    g_variant_unref(prop);
}
```

NOTE: FLOATING REFERENCES IN METHOD CALLS

`g_variant_new()` returns a *floating reference*. When passed to `g_dbus_proxy_call_sync()`, the library calls `g_variant_ref_sink()` to take ownership and frees it automatically. Never call `g_variant_unref()` on the parameters you pass into method calls.

6 Client-Side: Signal Subscriptions

6.1 Connection-Level Signal Subscription

To monitor broadcast signals directly on the connection, register a match rule with `g_dbus_connection_signal_subscribe()`:

```
static void on_result_computed(GDBusConnection *c,
                              const gchar *sender,
                              const gchar *path,
                              const gchar *iface,
                              const gchar *signal_name,
                              GVariant *params,
                              gpointer user_data)
{
    gint32 res = 0;
    g_variant_get(params, "(i)", &res);
    g_print("Signal received: ResultComputed(%d) from %s\n", res, sender);
}

guint sub_id = g_dbus_connection_signal_subscribe(
    conn,
    "com.example.AdderService",
    "com.example.Adder",
    "ResultComputed",
    "/com/example/Adder",
    NULL,
    G_DBUS_SIGNAL_FLAGS_NONE,
    on_result_computed,
    NULL,
    NULL
);
```

6.2 Handling PropertiesChanged Broadcasts

Standard D-Bus objects emit `PropertiesChanged` with signature `(sa{sv}as)` containing the interface name, a dictionary of updated attributes, and an array of invalidated names:

```
static void on_props_changed(GDBusConnection *c,
                             const gchar *sender,
                             const gchar *path,
                             const gchar *iface,
                             const gchar *signal_name,
                             GVariant *params,
                             gpointer user_data)
{
    const gchar *target_iface = NULL;
    GVariant *changed_dict = NULL;
    const gchar **invalidated = NULL;

    g_variant_get(params, "(&s@a{sv}^a&s)",
                  &target_iface,
                  &changed_dict,
                  &invalidated);

    GVariantIter iter;
    const gchar *key = NULL;
    GVariant *val = NULL;
    g_variant_iter_init(&iter, changed_dict);

    while (g_variant_iter_loop(&iter, "{&sv}", &key, &val)) {
        gchar *s = g_variant_print(val, TRUE);
        g_print("Property '%s' changed -> %s\n", key, s);
        g_free(s);
    }

    g_variant_unref(changed_dict);
    g_free(invalidated);
}
```

7 Server-Side: Low-Level VTable Implementation

7.1 Introspection XML Definition

A low-level server exports services by registering an introspection XML schema and a dispatch table:

```
static const gchar introspection_xml[] =
    "<node>"
    "  <interface name='com.example.Adder'>"
    "    <method name='Add'>"
    "      <arg type='i' name='a' direction='in'>"
    "      <arg type='i' name='b' direction='in'>"
    "      <arg type='i' name='result' direction='out'>"
    "    </method>"
    "    <property name='Enabled' type='b' access='readwrite'>"
    "  </interface>"
    "</node>";
```

7.2 Method Dispatch & Invocations

The handler receives a `GDBusMethodInvocation` token that *must* be completed with either a return value or an error:

```
static void handle_method_call(GDBusConnection *C,
                              const gchar *sender,
                              const gchar *path,
                              const gchar *iface,
                              const gchar *method,
                              GVariant *params,
                              GDBusMethodInvocation *invoc,
                              gpointer udata)
{
    if (g_strcmp0(method, "Add") == 0) {
        gint32 a = 0;
        gint32 b = 0;
        g_variant_get(params, "(ii)", &a, &b);

        GVariant *ret_val = g_variant_new("(i)", a + b);
        g_dbus_method_invocation_return_value(invoc, ret_val);
    } else {
        g_dbus_method_invocation_return_error(
            invoc,
            G_DBUS_ERROR,
            G_DBUS_ERROR_UNKNOWN_METHOD,
            "Unknown method: %s",
            method
        );
    }
}
```

7.3 Property Get/Set Handlers

```
static GVariant *handle_get_prop(GDBusConnection *c,
                                 const gchar *s,
                                 const gchar *p,
                                 const gchar *i,
                                 const gchar *prop,
                                 GError **err,
                                 gpointer u)
{
    if (g_strcmp0(prop, "Enabled") == 0) {
        return g_variant_new_boolean(TRUE);
    }
    g_set_error(err, G_DBUS_ERROR, G_DBUS_ERROR_UNKNOWN_PROPERTY, "Unknown property");
    return NULL;
}
```

NOTE: THE INVOCATION CONTRACT

8 Server-Side: Ownership & Signal Emission

8.1 VTable Linking & Object Registration

Combine handlers into a `GDBusInterfaceVTable` and export the object onto the active connection:

```
static const GDBusInterfaceVTable vtable = {
    handle_method_call,
    handle_get_prop,
    handle_set_prop
};

GDBusNodeInfo *node = g_dbus_node_info_new_for_xml(introspection_xml, &error);
guint reg_id = g_dbus_connection_register_object(
    conn,
    "/com/example/Adder",
    node->interfaces[0],
    &vtable,
    user_context,
    NULL,
    &error
);
g_dbus_node_info_unref(node);
```

8.2 Requesting a Well-Known Name

Acquire a service name across the bus using `g_bus_own_name_on_connection()`:

```
static void on_acquired(GDBusConnection *c, const gchar *name, gpointer u) {
    g_print("Successfully claimed bus name: %s\n", name);
}

static void on_lost(GDBusConnection *c, const gchar *name, gpointer u) {
    g_printerr("Lost ownership of bus name: %s\n", name);
}

guint owner_id = g_bus_own_name_on_connection(
    conn,
    "com.example.AdderService",
    G_BUS_NAME_OWNER_FLAGS_NONE,
    on_acquired,
    on_lost,
    NULL,
    NULL
);
```

8.3 Emitting Signals from the Server

Broadcast state changes to all listening clients:

```
// Broadcast ResultComputed(12)
GVariant *sig_params = g_variant_new("(i)", 12);

g_dbus_connection_emit_signal(
    conn,
    NULL,
    "/com/example/Adder",
    "com.example.Adder",
    "ResultComputed",
    sig_params,
    &error
);
```

TIP: AUTOMATIC SOCKET FLUSHING

Unlike legacy `libdbus`, `GDBus` queues outgoing messages directly inside its main context loop and flushes sockets automatically. You do not need to call `dbus_connection_flush()`. For immediate synchronous transmission, use `g_dbus_connection_flush_sync()`.

9 The Code Generation Way (gdbus-codegen)

9.1 Running gdbus-codegen

Writing manual vtables and GVariant packers is tedious. The official tool `gdbus-codegen` generates type-safe C proxies and server skeletons directly from XML schemas:

```
# Execute code generator from build system (Meson / Autotools / CMake)
gdbus-codegen \
  --generate-c-code adder-generated \
  --c-namespace Com \
  --interface-prefix com.example. \
  adder-interface.xml
```

9.2 Using the Generated Type-Safe Client Proxy

Call remote methods using native C types without constructing GVariants:

```
#include "adder-generated.h"

ComExampleAdder *proxy = com_example_adder_proxy_new_for_bus_sync(
    G_BUS_TYPE_SYSTEM,
    G_DBUS_PROXY_FLAGS_NONE,
    "com.example.AdderService",
    "/com/example/Adder",
    NULL,
    &error
);

gint32 sum = 0;
gboolean ok = com_example_adder_call_add_sync(proxy, 5, 7, &sum, NULL, &error);

if (ok) {
    g_print("Type-safe Add result: %d\n", sum);
}

gint32 last = com_example_adder_get_last_result(proxy);
g_object_unref(proxy);
```

9.3 Exporting the Generated Server Skeleton

Implement methods by connecting to standard GObject signals:

```
static gboolean on_handle_add(ComExampleAdder *iface,
                              GDBusMethodInvocation *invoc,
                              gint32 a,
                              gint32 b,
                              gpointer udata)
{
    com_example_adder_complete_add(iface, invoc, a + b);
    return TRUE;
}

ComExampleAdder *skeleton = com_example_adder_skeleton_new();
g_signal_connect(skeleton, "handle-add", G_CALLBACK(on_handle_add), NULL);

g_dbus_interface_skeleton_export(
    G_DBUS_INTERFACE_SKELETON(skeleton),
    conn,
    "/com/example/Adder",
    &error
);
```

TIP: AUTOMATIC PROPERTIESCHANGED EMISSION

When using skeletons, calling setter functions like `com_example_adder_set_last_result(skeleton, 42);` updates internal state and **automatically emits PropertiesChanged** across the bus. You never have to build `sa{sv}` as variants by hand.

10 GMainLoop, Async Operations & Deferred Replies

10.1 Asynchronous Client Method Calls

To avoid blocking the UI thread or event loop during network/IPC latency, make non-blocking calls:

```
static void on_add_done(GObject *src, GAsyncResult *res, gpointer u)
{
    GError *err = NULL;
    GVariant *val = g_dbus_proxy_call_finish(G_DBUS_PROXY(src), res, &err);

    if (val) {
        gint32 sum = 0;
        g_variant_get(val, "(i)", &sum);
        g_print("Async returned: %d\n", sum);
        g_variant_unref(val);
    }
}

g_dbus_proxy_call(
    proxy,
    "Add",
    g_variant_new("(ii)", 5, 7),
    G_DBUS_CALL_FLAGS_NONE,
    -1,
    NULL,
    on_add_done,
    NULL
);
```

10.2 Deferred Method Replies (Async Server Operations)

When fulfilling a request requires slow hardware I/O, keep the invocation alive with `g_object_ref()` and reply later:

```
static void handle_slow_op(GDBusMethodInvocation *invoc)
{
    GDBusMethodInvocation *pending = g_object_ref(invoc);
    start_hardware_operation(on_hardware_done, pending);
}

static void on_hardware_done(gboolean ok, gint32 val, gpointer udata)
{
    GDBusMethodInvocation *pending = udata;

    if (ok) {
        GVariant *ret_val = g_variant_new("(i)", val);
        g_dbus_method_invocation_return_value(pending, ret_val);
    } else {
        g_dbus_method_invocation_return_error(
            pending,
            G_DBUS_ERROR,
            G_DBUS_ERROR_FAILED,
            "HW Error"
        );
    }

    g_object_unref(pending);
}
```



Figure 10.1: Server execution lifecycle during non-blocking deferred method execution.

11 Credential Checking & Security Verification

11.1 Inspecting Caller UID at Runtime

For administrative operations (e.g. factory resets, power actions), verify the caller's OS credentials by querying the broker:

```
static void handle_factory_reset(GDBusMethodInvocation *invoc)
{
    const gchar *caller = g_dbus_method_invocation_get_sender(invoc);
    GDBusConnection *conn = g_dbus_method_invocation_get_connection(invoc);
    GError *err = NULL;

    GVariant *caller_var = g_variant_new("(s)", caller);
    GVariant *res = g_dbus_connection_call_sync(
        conn,
        "org.freedesktop.DBus",
        "/org/freedesktop/DBus",
        "org.freedesktop.DBus",
        "GetConnectionUnixUser",
        caller_var,
        G_VARIANT_TYPE("(u)"),
        G_DBUS_CALL_FLAGS_NONE,
        -1,
        NULL,
        &err
    );

    guint32 uid = 0;
    g_variant_get(res, "(u)", &uid);
    g_variant_unref(res);

    if (uid != 0) {
        g_dbus_method_invocation_return_error(
            invoc,
            G_DBUS_ERROR,
            G_DBUS_ERROR_ACCESS_DENIED,
            "Root required (UID=%u)",
            uid
        );
        return;
    }

    do_reset_sequence();
    g_dbus_method_invocation_return_value(invoc, NULL);
}
```

11.2 Credential Inspection Interfaces

Method / API Call	Return Type	Purpose & Security Usage
GetConnectionUnixUser	uint32 (u)	Retrieves the numeric POSIX UID of the caller.
GetConnectionUnixProcessID	uint32 (u)	Returns the PID of the calling client process.
GetConnectionCredentials	a{sv}	Returns rich dictionary (UID, PID, SELinux security context).
g_dbus_connection_get_peer_credentials()	GCredentials*	Used exclusively for peer-to-peer (P2P) connections.

WATCH OUT: TWO-TIER SECURITY ARCHITECTURE

Never rely exclusively on application-level UID checks. The primary line of defense is always the **XML policy configuration** in `/etc/dbus-1/system.d/`. The runtime check acts as an internal safeguard.

EXTRA: COMPARISON WITH SD-BUS CREDENTIALS

In `sd-bus`, inspecting credentials is significantly simpler via `sd_bus_query_sender_creds()`, which reads metadata directly from kernel socket `SO_PEERCREC` without an extra D-Bus round-trip query.

12 Memory Rules, Comparison & Quick Reference

12.1 The 6 Golden Memory Rules of GDBus

Entity	Allocation & Acquisition Source	Disposal & Ownership Contract
GObject	<code>g_bus_get_sync</code> , <code>g_dbus_proxy_new_sync</code>	Balance every reference with exactly one <code>g_object_unref()</code> .
GVariant Calls	<code>g_dbus_proxy_call_sync()</code> results	Caller owns the return value; must call <code>g_variant_unref()</code> .
Floating Ref	<code>g_variant_new("(ii)", 5, 7)</code> passed to call	Consumed by callee (floating sink). Do NOT unref.
Strings	<code>g_variant_get(v, "s", &str)</code> vs <code>"&s"</code>	"s" requires <code>g_free(str)</code> ; "&s" is borrowed, do not free.
GError	Initialized to NULL, filled on failure	Free with <code>g_error_free()</code> or reset with <code>g_clear_error()</code> .
Iterators	<code>g_variant_iter_loop()</code>	Loop manages sub-items; do not manually free inside loop.

12.2 Library Comparison: libdbus vs. sd-bus vs. GDBus

Feature Metric	libdbus-1 (Legacy)	sd-bus (systemd)	GDBus (GLib / GIO)
Header / Library	<code>dbus/dbus.h</code> (libdbus-1)	<code>systemd/sd-bus.h</code> (libsystemd)	<code>gio/gio.h</code> (libgio-2.0)
Main Loop	Manual socket polling	Native sd-event (or epoll)	GMainLoop / GMainContext
Type System	Manual message iterators	Type format strings	GVariant architecture
Codegen Tool	None	None	gdbus-codegen
Property Caching	No	No	Yes (Automatic in GDBusProxy)
Primary Use Case	Legacy codebases	Embedded daemons & PID 1	Modern Linux desktop & GLib daemons

12.3 Diagnostic Tooling Reference

G_DBUS_DEBUG=all

Environment flag that prints full protocol traces, SASL authentication states, and raw byte stream payloads directly to stderr.

busctl (CLI Tool)

Inspect running services: `busctl tree <dest>`, `busctl introspect <dest> <path>`, and `busctl call` to test endpoints rapidly.

Linux Systems Architecture Series: HARDIK — www.linkedin.com/in/hardikhardik