Reliable and Efficient Dual-OS Communications for Real-Time Embedded Virtualization

Daniel Sangorrín, Shinya Honda, Hiroaki Takada

Dual-OS communications allow a real-time operating system (RTOS) and a general-purpose operating system (GPOS)—sharing the same processor through virtualization—to collaborate in complex distributed applications. However, they also introduce new threats to the reliability (e.g., memory and time isolation) of the RTOS that need to be considered. Traditional dual-OS communication architectures follow essentially the same conservative approach which consists of extending the virtualization layer with new communication primitives. Although this approach may be able to address the aforementioned reliability threats, it imposes a rather big overhead on communications due to unnecessary data copies and context switches.

In this paper, we propose a new dual-OS communications approach able to accomplish efficient communications without compromising the reliability of the RTOS. We implemented our architecture on a physical platform using a highly reliable dual-OS system (SafeG) which leverages ARM TrustZone hardware to guarantee the reliability of the RTOS. We observed from the evaluation results that our approach is effective at minimizing communication overhead while satisfying the strict reliability requirements of the RTOS.

1 Introduction

Modern high-end embedded systems are no longer confined to standalone, resource-constrained devices. On the contrary, they frequently consist of highly connected devices with increasing sophistication and complexity. A promising approach to cope with such complexity is the dual-OS approach [1] [2] [3] which is depicted in Fig. 1. A dual-OS system consists of a real-time operating system (RTOS) and a general-purpose operating system (GPOS) running on the same processor—to reduce the hardware cost—thanks to the use of a virtualization layer (VL). While the RTOS and the virtualization layer are considered to belong to the trusted computing base (TCB), the GPOS is considered to belong to the untrusted computing base (UCB) due to its large scale. For that reason, the most fundamental requirement for a dual-OS system is to protect the reliability (e.g., memory and time isolation) of the TCB against the UCB [4] [5].

Although the mere execution of the RTOS and the GPOS in isolation may satisfy the requirements of some systems, support for communication between both OSs opens the door for new applications with higher sophistication. Despite its many benefits, dual-OS communications introduce new threats to the reliability of the RTOS that must also be considered. Some existing dual-OS communication systems [2] [6] [7] [8] are able to cope with most of the aforementioned reliability threats. However, they all essentially follow the same traditional ap-
Fig. 2 Architecture of the SafeG dual-OS system.

proach (see Fig. 6a) based on extending the dual-OS virtualization layer with additional communication primitives. Although the traditional approach simplifies synchronizing both OSs and protecting communication structures, it has several efficiency drawbacks to consider, such as the use of unnecessary data copies and context switches.

The main contribution of this work is a new dual-OS communications architecture able to provide efficient communications without compromising the reliability of the RTOS. We implemented it on a physical platform using a highly reliable dual-OS system called SafeG. Our implementation leverages ARM TrustZone hardware security extensions to guarantee the memory protection and timeliness of the RTOS at a rather low overhead. We evaluated our approach through several experiments; and compared the results with the traditional approach used in previous dual-OS systems. From the evaluation results, we observed that the proposed architecture is indeed effective at minimizing the communication overhead while satisfying the strict reliability requirements of the RTOS.

The remainder of this paper is structured as follows. Section 2 reviews background knowledge for better understanding the contents of this paper. We also introduce related work and detail the efficiency and reliability problems that we encountered in previous approaches to dual-OS communications. Section 3 provides a list of requirements and assumptions for the design of our dual-OS communications architecture. Section 4 constitutes the core of this paper and describes our approach to dual-OS communications. Section 5 presents the results of the evaluation of our reference implementation and discusses the satisfaction of the requirements listed in Section 3. Finally, Section 6 presents our concluding remarks.

2 Background

2.1 Review of SafeG

Fig. 2 depicts the architecture of SafeG (Safety Gate), a reliable and open-source dual-OS system based on ARM TrustZone hardware. Here, we briefly introduce some concepts about SafeG and TrustZone. For details, refer to [1][9][11][12].

- Virtual CPUs: a processor core contains two Virtual CPUs (VCPUs), the Secure and the Non-Secure VCPU. VCPUs are executed in a time-sliced fashion (i.e., not in parallel). Each VCPU is equipped with its own memory management unit (MMU) and exception vectors; and supports all ARM operation modes. SafeG assigns the RTOS and GPOS to the Secure and Non-Secure VCPUs respectively.

- SafeG monitor: the Secure VCPU has an additional mode—called monitor mode—which is used by the SafeG monitor to context switch between both OSs. The entry to SafeG monitor can only be triggered by software executing the Secure Monitor Call (SMC) instruction or the occurrence of an FIQ (Fast Interrupt Request) while the Non-Secure VCPU is active. The SafeG monitor is small—around 2KB—and executes with all interrupts disabled, which simplifies its verification. A VCPU context switch on an ARM1176 core requires around 200 cycles and involves saving and restoring all ARM general-purpose registers.

- Address space partitioning: when a bus master
accesses memory or devices, the NS bit (Non-Secure bit) is propagated through the system bus indicating the privilege of that access (i.e., secure or non-secure). This allows partitioning the address space into two virtual worlds: the Secure and the Non-Secure world. The Secure VCPU can access memory and devices from both worlds. However, hardware makes sure that Secure memory and devices cannot be accessed by the Non-Secure VCPU or Non-Secure DMA devices. At initialization, SafeG configures RTOS memory and devices as Secure resources; and GPOS memory and devices as Non-Secure resources. For that reason, the RTOS address space is protected against malicious accesses from the untrusted GPOS.

- **Interrupts partitioning:** ARM processors have two types of interrupt known as FIQ and IRQ. The main difference is that FIQs have higher priority and more banked registers. SafeG configures RTOS devices to generate FIQs; and GPOS devices to generate IRQs. This configuration is done through a TrustZone Interrupt Controller (TZIC), only accessible from the Secure VCPU. FIQs and IRQs can be disabled in privileged mode by setting the F and I flags of the Current Program Status Register (CPSR) respectively. To prevent the GPOS from masking RTOS device interrupts, SafeG takes advantage of the FW (F flag Writable) bit, which is only accessible by the Secure VCPU. This allows the RTOS to ensure that hard real-time tasks always meet their deadlines.

The execution flow of the SafeG architecture is controlled by the two paths depicted in Fig. 3:

- **PATH 1 (SMC)** is used for the RTOS to initiate a context switch to the GPOS through the SMC instruction. The instant at which the RTOS initiates a context switch depends on

  - the scheduling algorithm. SafeG supports two scheduling algorithms. The most basic algorithm (see Fig. 4a) is *idle scheduling* [1] which guarantees that the GPOS only executes when the RTOS is idle. A more complex scheduling algorithm (see Fig. 4b) is *integrated scheduling* [14], which supports mixing the global priority level of RTOS and GPOS activities in a reliable way by assigning scheduling servers (i.e., an execution budget, replenishment period and foreground priority) to GPOS activities.

- **PATH 2 (FIQ)** occurs when an FIQ interrupt arrives to the processor while the Non-Secure VCPU is active. The arrival of the FIQ interrupt forces the processor to enter Monitor mode, where the FIQ vector handler of the SafeG monitor switches back to the RTOS.

### 2.2 Dual-OS communications

A dual-OS communications system is a set of methods for the exchange of information between RTOS and GPOS applications—running on a dual-OS system—which opens the possibilities for new applications with higher sophistication:

- **RTOS ⇒ GPOS communications** are typically used for the RTOS to leverage the rich functionality of the GPOS. Fig. 5 shows an example where dual-OS communications are used
for the RTOS to report the status of a group of heating devices to the GPOS, which is in turn connected to a remote user through a network.

- **GPOS ⇒ RTOS** communications are typically used for the GPOS to request a service that only the RTOS can provide. For example, the RTOS may provide security services [12][9] (e.g., digital rights or authentication services) that use cryptographic keys stored on secure memory that are not accessible by the GPOS. Another application is device sharing [15][16][17] which cannot be accomplished through GPOS devices—even if both OSs have access to them—because otherwise the GPOS could access sensible data belonging to the RTOS; or change the device's configuration.

### 2.3 Related work

In spite of its benefits, dual-OS communications also introduce new threats to the reliability of the RTOS: message overload attacks [18]; user and control data corruption attacks [19]; memory faults caused by shared pages being removed [20]; or unbounded waits caused by the non-cooperation of the GPOS are just a few examples.

There already exist several dual-OS communication systems [2][6][7][8] capable of addressing most of the aforementioned reliability threats. However, they all essentially follow the same rather conservative traditional approach (see Fig. 6a) that requires the communications data path to traverse the VL, which needs to be extended with additional communication primitives. This approach simplifies the synchronization between both OSs—typically by disabling interrupts inside the VL—in the access to the control data used for communications; and is useful to protect it against GPOS attacks.

However, it also has several efficiency drawbacks that must be considered, such as unnecessary data copies and context switches [21][22].

In parallel, several works [23][24][25] have addressed the demanding efficiency requirements of inter-OS communications for multiple-guest hypervisors (e.g., XEN [26]) used in enterprise cluster computing. Most of them exploit the use of kernel shared memory for reducing the number of data copies (see Fig. 6b). Unfortunately, none of these hypervisors are able to guarantee the high reliability requirements of a dual-OS system.

### 3 Requirements and assumptions

This section presents a list of requirements and assumptions for the design of a reliable and efficient dual-OS communications system. The satisfaction of these requirements will be discussed in Section 5.

#### 3.1 Reliability requirements

(a) **Memory isolation**: TCB memory must be protected against any access—accidental or malicious—by the GPOS. Shared memory used for communications must only be accessible by the RTOS and GPOS privileged software, which should pass a software quality control for increasing its trustworthiness.

(b) **Shared control data**: control data shared by both OSs must be validated (e.g., using range checking) by the RTOS, and protected against further malicious modifications by the GPOS.

(c) **Real-time**: the timeliness of RTOS interrupt handlers and tasks must be guaranteed. In particular, it must be protected against message overload attacks coming from the GPOS.

(d) **Memory faults**: the architecture must guarantee that the RTOS will not access non-existent memory—causing the corresponding memory
Table 1 Requirements Vs. Our design choices.

<table>
<thead>
<tr>
<th>Type</th>
<th>Requirement</th>
<th>Number</th>
<th>Evaluation</th>
<th>Our design choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>Memory isolation</td>
<td>3. 1a</td>
<td>QL</td>
<td>Three trustworthiness levels.</td>
</tr>
<tr>
<td></td>
<td>Shared control data</td>
<td>3. 1b</td>
<td>QL/QN</td>
<td>Four steps update algorithm.</td>
</tr>
<tr>
<td></td>
<td>Real-time</td>
<td>3. 1c</td>
<td>QN</td>
<td>Message interrupt rate limiting.</td>
</tr>
<tr>
<td></td>
<td>Memory faults</td>
<td>3. 1d</td>
<td>QL</td>
<td>Reserve shared memory at configuration.</td>
</tr>
<tr>
<td></td>
<td>Unbounded blocking</td>
<td>3. 1e</td>
<td>QL</td>
<td>Timeouts and non-blocking interface.</td>
</tr>
<tr>
<td></td>
<td>Code modifications</td>
<td>3. 1f</td>
<td>QL/QN</td>
<td>User-level library based on the RTOS API.</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Throughput</td>
<td>3. 2a</td>
<td>QN</td>
<td>Shared memory; filters; non-synchronized.</td>
</tr>
<tr>
<td></td>
<td>Memory size</td>
<td>3. 2b</td>
<td>QN</td>
<td>Static configuration interface.</td>
</tr>
<tr>
<td></td>
<td>Interface</td>
<td>3. 2c</td>
<td>QL</td>
<td>Shared memory &amp; events; RPCs; unqueued or sampling messages.</td>
</tr>
</tbody>
</table>

QL=Qualitative QN=Quantitative

Faults—even if GPOS tasks are swapped out to virtual storage or shared memory used for communications is unmapped by the GPOS.
(e) Unbounded blocking: the architecture must guarantee that RTOS tasks will not suffer unbounded waiting times caused by the non-cooperation of the GPOS.
(f) Code modifications: the architecture must not impose modifications to the RTOS kernel or the VL to avoid reissuing a new verification process, and for improving the maintainability of the architecture. The GPOS kernel can be extended with a driver (e.g. to handle communication events) but its core code must not be modified due to maintenance reasons.

3.2 Efficiency requirements
(a) Throughput: the architecture must minimize the overhead caused by unnecessary data copies and context switches; and the use of costly protocol stacks.
(b) Memory size: the amount of memory used for dual-OS communications must be minimized.
(c) Interface: the communications interface must be suitable for implementing communication patterns—such as remote procedure calls (RPC) or unqueued communication—typically present in common embedded systems.

3.3 Assumptions
(a) Static: all RTOS communication resources can be statically allocated for reliability reasons.
(b) Transparency: the user interface does not require being transparent to the user.
(c) Events driver: the GPOS allows implementing a driver to send or receive software interrupts.
(d) Raw user data: validating the raw contents of user messages is out of the scope of this paper.
(e) Verified TCB: we assume that software belonging to the TCB has been correctly verified and does not have any defects.

4 Communications architecture
This section describes our dual-OS communications architecture (hereafter dualoscom architecture). Table 1 summarizes our design choices for the list of requirements presented in Section 3. In Section 5 we evaluate whether these design choices are able to satisfy the mentioned requirements or not; and compare their performance against the traditional approach depicted in Fig. 6a.

4.1 Satisfying reliability requirements
- Memory isolation (req. 3.1a): the dualoscom architecture relies on user-level shared memory for implementing dual-OS communications efficiently (see Fig. 7). In contrast to the traditional approach depicted in Fig. 6a, dualoscom control data structures are not protected by the VL. Instead, we divide GPOS tasks into two groups: GPOS communicating tasks with the privilege of accessing the shared memory region; and other GPOS tasks without such privilege. GPOS communicating tasks are created and thoroughly tested by the dual-OS system engineer during the development phase.
In contrast, the other GPOS tasks may include malicious or buggy applications installed by the user during the lifetime of the system, and are expected to be less trustworthy. This reasoning leads to the existence of three trustworthiness levels (trusted, untrusted-privileged and untrusted-unprivileged), which are separated by the two protection sandboxes illustrated by Fig. 7. TCB memory is protected against any GPOS access by the VL protection level; and communications shared memory is protected against untrustworthy GPOS tasks by the permissions-based protection level.

- **Shared control data** (req. 3.1b): if the permissions-based protection level is broken (e.g., by exploiting a GPOS kernel bug), untrusted-unprivileged GPOS tasks can attack the second virtualization-based protection level by maliciously modifying the shared control data. In order to protect the RTOS against such modifications—for example, to avoid dereferencing a null pointer—all updates by the RTOS to the shared control data are made in four steps: copy the required control data to the RTOS memory; validate it by range-checking; update it according to the current operation (e.g., enqueue); and finally, copy the modified control data back to shared memory. Validating the raw contents of user messages is out of the scope of this paper (see assumption 3.3d).

- **Real-Time** (req. 3.1c): another way for a malicious GPOS application to attempt breaking the second protection level is by sending an excessive amount of messages to the RTOS. The dualoscom architecture splits the transmission of messages in two parts: the data path and the events path. The data path involves non-blocking operations to enqueue or dequeue blocks of data; and is implemented through lock-free bidirectional FI-FOs that exist in shared memory (see Fig. 8). The events path involves asynchronous notifications (implemented through inter-OS interrupts) and wait-event operations that may block the calling task until a timeout expires. This separation allows tasks to communicate using both polling or event-driven communication patterns. However, in order to protect the timeliness of RTOS activities the rate of GPOS ⇒ RTOS message interrupts must be limited. The dualoscom architecture supports two message interrupt limiters: the strict message interrupt limiter which enforces the minimum inter-arrival time between interrupts; and the bursty message interrupt rate limiter which enforces a maximum burst size and a maximum arrival rate[18]. Finally, it is the responsibility of RTOS applications not to poll for new GPOS messages in an endless loop.

- **Memory faults** (req. 3.1d): to guarantee that the RTOS will never try to access non-existent or unmapped memory, the shared memory region used for communications is statically allocated at configuration time.

- **Unbounded blocking** (req. 3.1e): to avoid a situation in which RTOS tasks could wait for a GPOS message for an unbounded amount of time, a timeout can be specified in all blocking
operations of the events path. Non-blocking operations never perform retries, and return an error code instead when there is contention.

- **Code modifications** (req. 3.1f): to avoid modifying the RTOS kernel or the VL, the data path is carried out by the dualoscom communications library (comm. lib) at user level. The events path and the message interrupt rate limiting mechanisms are implemented through the RTOS application interface (API). In contrast, to implement event operations (e.g., waiting or sending an event), the GPOS kernel requires being extended with a communications driver.

### 4.2 Satisfying efficiency requirements

- **Throughput** (req. 3.2a): to minimize the overhead caused by unnecessary data copies and context switches, all data path communications occur at user level through shared memory. To reduce the overhead caused by the events path, applications can choose to use a single event to notify the transmission of several messages, thus reducing the number of context switches per message. This is supported by splitting the communications interface between the transmission of data and events. There are two more mechanisms for reducing the overhead caused by unnecessary context switches: filters and non-synchronized accesses. Filters are functions that execute on the sender side of a channel (see Fig. 8 and Fig. 9) and are used for discarding the transmission of messages when they are not needed by the receiver (e.g., if a variable has not changed since the last time it was received). The access to a channel can be configured to be synchronized (e.g., through a mutex supporting the priority ceiling or the priority inheritance algorithm) or non-synchronized. This allows avoiding the execution time overhead associated to access synchronization when only a single task is supposed to access the channel.

- **Memory size** (req. 3.2b): to minimize the amount of memory used by dual-OS communications, all channel parameters can be configured. The configuration parameters of a channel include: the number of blocks and their size; the use of synchronized accesses; and its associated filters.

- **Interface** (req. 3.2c): the run time interface to the dualoscom architecture supports shared memory blocks and asynchronous event notifications. By combining them it is possible to build more complex communication patterns such as RPCs or unqueued messages [28] [6]. See Section 4.4.3 and Section 4.5 for details.

### 4.3 Communication channels

A channel is a communication entity by means of which RTOS and GPOS untrusted-privileged tasks can exchange information. Fig. 8 depicts the main structures of a communication channel, which is composed of the following elements:

- **Blocks**: a block is a piece of shared memory used to send data. Each channel contains a pool of a configurable number of blocks. All the blocks in a channel have a fixed size, which is also configurable. Blocks must be explicitly allocated before being used. They can be sent in both directions (i.e., RTOS \(\rightarrow\) GPOS) and they can be released back to the channel’s pool either by the sender or the receiver.

- **FIFOs**: a FIFO (First-In-First-Out) queue is a data structure used to deliver blocks in the same order they were enqueued. Each channel contains exactly two FIFOs, one for each communication direction. A FIFO has a number of elements equal to the number of blocks in the
channel. Each enqueued element consists of a block identifier that was previously allocated and enqueued by a sender. A FIFO queue can be easily implemented using a lock-free algorithm if all of its operations are serialized. For that reason, the GPOS does not need to disable RTOS interrupts (e.g., for synchronization purposes) which is forbidden by the VL.

- **Filters**: a filter is a function that receives a block’s buffer and size, and returns a boolean to indicate whether the corresponding block should be sent or not. Filters are used for discarding the transmission of a block (i.e., before it is enqueued) depending on its contents. They are used to avoid unnecessary communication overhead [28]. For example, in Fig. 10 a filter function (tmp_update) is used to discard the transmission of heater temperature values (see Fig. 5) that do not represent updates of previous values. Fig. 9 depicts the dualoscom filtering functionality. Each channel contains two active filter functions (e.g., rtfilter2 and gpfilter0), one for each communication direction. Filters used in the RTOS ⇒ GPOS communication direction (e.g., rtfilter#) execute on the RTOS, and therefore must follow the same formal verification process as other components in the TCB (e.g., they must assume that blocks can be maliciously modified by the GPOS). In contrast, GPOS ⇒ RTOS filters (e.g., gpfilter#) execute on the GPOS untrusted-privileged user space. Compared to untrusted-unprivileged software, GPOS filters must follow a software quality control. However, they are allowed to assume that data sent by the RTOS is valid. The source code of RTOS and GPOS filters is statically provided by the dual-OS system engineer during the build process (see Fig. 10), and their contents cannot be modified during the execution of the system. Instead, each channel contains two variables (RTOS and GPOS active filter id), which are identifier numbers for indicating the currently active filter on each communication direction (e.g., 2 in Fig. 9a and 0 in Fig. 9b). While filter functions are located in the same memory region as the operating system where they execute, active filter id variables are located in shared memory. Receiver tasks can select the active filter at run time by using a filter identifier—or a null value if no filtering is required—as illustrated by Fig. 9. Filter identifiers are automatically allocated by the dualoscom configurator tool during the configuration phase (see Fig. 10), and are internally represented as natural integers. Before a block is enqueued to a channel, the dualoscom library reads its active filter id number from shared memory, and executes the associated active filter function (e.g., rtfilter2 and gpfilter0) on it. If the filter function returns true, the block is enqueued to the FIFO; otherwise an error code is returned to the user, indicating that the block was discarded. Note that in Fig. 9a, a malicious GPOS task can potentially corrupt the active filter id with an out-of-range value (e.g., 47). For that reason, the RTOS library must always validate the range of the active filter id variable before using it to select a filter function for execution.

- **Events**: an event is a method for sending asynchronous notifications between the RTOS and the GPOS. Events can be sent in both directions and they are not queued, meaning that they must be acknowledged by the receiver before a new event can be sent. Events are
sent independently to the process of enqueueing blocks. This allows senders to enqueue several blocks before notifying the receivers.

- **Mutexes**: A mutex is a mechanism used for serializing the access of tasks to a channel within the same OS. Channels can have up to two mutexes, one for each communication direction. Each mutex can be removed at configuration time—for minimizing the synchronization overhead—if access contention is not expected.

4.4 Dualoscom interface

4.4.1 The dualoscom build process

Fig. 10 illustrates the dualoscom build process through the heating devices example from Fig. 5. As it is common practice in most RTOSs[10], dualoscom provides a configuration interface which allows all of its structures to be allocated statically. This is necessary for guaranteeing the reliability of the TCB and it allows minimizing its memory and execution time overhead. First, the dual-OS system engineer provides a configuration file (e.g., `dualoscom_config.txt`) containing a channel declaration for each heating device. Its syntax is detailed in Section 4.4.2. Then, the configuration file is parsed by the `dualoscom configurer` tool, which generates a configured header file (e.g., `dualoscom_config.h`) with constant definitions (e.g., identifiers); and an RTOS configuration file (e.g., `rtos.cfg`) with static declarations. Normally RTOS resources (e.g., semaphores) are allocated statically for reliability reasons[27]. Next, the dual-OS system engineer provides the RTOS and GPOS communicating applications, and the filter functions if necessary. Applications use the run time interface in Section 4.4.3 for communicating.

The build process ends with the generation of two binaries: the RTOS bare-metal binary (e.g., `asp.bin`), which contains the RTOS kernel, dualoscom library and application (the RTOS kernel is typically linked to the user application for performance reasons); and the GPOS user application (e.g., `user.elf`) which is linked to the dualoscom library. The GPOS kernel is patched with...
a dualoscom driver, which receives all configuration parameters from user-space at initialization, and therefore it only needs to be built once.

4.4.2 Configuration interface

The configuration interface uses the next syntax:

DUALOSCOM_FILTER(): used to declare a filter. It accepts the following parameters:
- FILTER_NAME: a name for the filter. The dualoscom configurator generates a constant with the same name (e.g., TMP_FILTER) which is used as an identifier for the receiving tasks to select the active filter at run time.
- filter: the name of a function which takes a block's buffer and size as input parameters, and returns a boolean value (see the tmp_update function in Fig. 10 for an example). If the return value is true, the block will be enqueued; otherwise it will be discarded. The body of the function is written and tested (i.e., it must follow the same software quality controls as any other software executed in the same trustworthiness level) by the dual-OS system engineer before the build process (see Fig. 10) begins.

DUALOSCOM_CHANNEL(): used to declare a channel. It accepts the following parameters:
- CHANNEL_NAME: a name for the channel. After configuration, the same name (e.g., CH_A) can be used to identify the channel.
- num_blocks: the number of blocks.
- block_size: the block size in memory words.
- mutexes: two booleans to indicate if mutual exclusion is used at each communication end.
- rtos_filters: list of FILTER_NAME values to declare which filters can be selected on the RTOS end of this channel. The value NULL can be used to declare no filter. By default, there is no active filter at initialization.
- gpos_filters: list of FILTER_NAME values to declare which filters can be selected on the GPOS end of this channel. The value NULL can be used to declare no filter. By default, there is no active filter at initialization.

DUALOSCOM_CHANNELS_GROUP(): used to declare a group of channels, to allow waiting for events on several channels at the same time.
- GROUP_NAME: a name for the group of channels. After configuration, the same name can be used to identify the group.
- channels: a list of CHANNEL_NAME values that must match the values used during the declaration of channels.

4.4.3 Run time interface

The run time interface is a set of functions for the RTOS and GPOS applications to communicate between each other at run time. All functions return DUALOSCOM_SUCCESS upon success and one of the following errors upon failure:

1. DUALOSCOM_NOPERM: not enough permissions.
2. DUALOSCOM_NOINIT: the communications system is not initialized yet.
3. DUALOSCOM_PARAM: incorrect parameter.
4. DUALOSCOM_FULL: there are no free blocks.
5. DUALOSCOM_ENQ: the block is enqueued.
6. DUALOSCOM_FILTER: the block was discarded
7. DUALOSCOMEMPTY: no block is enqueued.
8. DUALOSCOMALLOC: the block is not allocated.
9. DUALOSCOM_TIMEOUT: a timeout occurred.

The run time interface is composed of the following list of functions. Note that the prefix dualoscom_ has been omitted from each function for the sake of shortness.

**Initialization functions**
- init(timeout): initializes the dualoscom system. The initialization protocol and the timeout units are implementation-dependent. May return errors 1, 3, and 9.

**Block management functions**
- block_alloc(chan_id, &block_id): it allocates a block from a channel's pool. This function never blocks the calling task. May return errors 1, 2, 3, and 4.
- block_free(block_id): releases a block back to the channel's pool where it belongs. May return errors 1, 2, 3, and 8.
- block_getbuffer(block_id, &buffer_p): to obtain a pointer to the beginning of the memory region of a block. May return errors 1, 2, 3, and 8.
- block_enqueue(block_id): enqueues a block to a channel's FIFO. Note that the channel is implicit in the block identifier. May return errors 1, 2, 3, 6, and 8.
- block_dequeue(chan_id, &block_id): dequeues a block from a channel's FIFO. This function never blocks the calling task. May return er-
Event management functions

- **event_send(channel_id)**: sends a channel event notification. If a notification had already been sent but not acknowledged by the receiver, it returns DUALOSCOM_SUCCESS. Otherwise it may return errors 1, 2, and 3.

- **event_wait(chan_id, timeout)**: this function makes the calling task wait for an event notification on a channel. If an event was pending, the function acknowledges it and returns immediately. Otherwise, the calling task is put in waiting state until an event arrives or a timeout occurs. The timeout units are implementation-dependent. May return errors 1, 2, 3, and 9.

- **event_select(group_id, &chan_id, timeout)**: this function makes the calling task wait for an event notification on a specific group of channels at the same time. If an event on one of the channels was pending, the function acknowledges it and returns immediately. Otherwise, the calling task is put in waiting state until an event arrives or a timeout occurs. The timeout units are implementation-dependent. May return errors 1, 2, 3, and 9.

Filter management functions

- **filter_set(chan_id, filter_id)**: used by receiver tasks to select one of the filter functions available at the sending side of a channel through a filter identifier. The filter identifier can be NULL_FILTER if no filtering is desired. May return errors 1, 2, and 3.

4.5 Middleware

This section describes an example implementation of remote procedure calls (RPCs) and unqueued messages [28][6] using the dualoscom interface.

4.5.1 RPC communication

Dual-OS RPC communications allow an RTOS client to request the execution of a subroutine by a GPOS server (or vice versa) in the same manner as if the subroutine was local. Fig. 11 outlines the pseudocode of a simple algorithm—error checking is not shown—for accomplishing RPC communications on top of the basic dualoscom interface. RPCs are internally implemented through client request messages sent over dualoscom channels (e.g., RPC_ADD). Each request message contains the input parameters (e.g., a and b) for the subroutine, and memory space for the RPC server to store the output parameters (e.g., result). If the RPC is synchronous, the client is put into waiting state while the server processes the request message. From the client point of view, the fact that the subroutine (e.g., add) executes remotely is completely transparent. The only difference compared to a local subroutine is the fact that the RPC must be declared on a configuration file (e.g., rpc_config.txt) through the following syntax:

DUALOSCOM_RPC(): used to declare an RPC.

- **function**: the name of the function.
- **direction**: indicates the communication direction (RPCs are unidirectional).
- **mutex**: indicates if mutual exclusion is required at the client end.
- **params**: a list of parameters with the next format: `param : [in] [out] type`.

The RPC configuration file is parsed by the RPC
Unqueued/Sampling messages

4.5.2 Unqueued/Sampling messages

Unqueued messages[28]—also known as sampling messages—are a useful method for the RTOS tasks to share data samples in a loosely-coupled fashion with the GPOS tasks. A data sample consists of a typically small region of memory containing a value that is updated periodically by a producer task. This value is read periodically by a loosely-coupled set of consumer tasks. Unqueued messages are useful for situations in which only the last value of some data (e.g., sensor data) is relevant to the application. Fig. 12a illustrates a simple way to implement unqueued messages on top of the dualoscom architecture. The example consists of two data samples sent in the RTOS \( \Rightarrow \) GPOS communication direction. Data samples (e.g., S1 and S2) are declared in a configuration file (e.g., samp_config.txt) using the following syntax:

\[
\text{DUALOSCOM\_SAMPLE()}: \text{used to declare a data sample.}
\]

- **SAMPLE\_NAME**: the name of the data sample.
- **mx\_size**: the maximum size of the sample.
- **direction**: indicates the communication direction (e.g., RTOS \( \Rightarrow \) GPOS).
- **mutex**: indicates if mutual exclusion is required to allow having multiple producers for the same data sample.
- **filter**: default filter.
- **init**: data sample initialization function.

The samples configuration file is parsed by the *samples configurator* tool (see Fig. 12b). This tool generates a sampling header (e.g., *samp\_config.h*) that contains the definition of several constants (e.g., the sample identifiers); and the dualoscom configuration file (e.g., *dualoscom\_config.txt*), which contains a channel declaration per data sample (e.g., \texttt{SAMP\_S1} and \texttt{SAMP\_S2}) and a group. RTOS producer tasks periodically send new sample values through these channels. On the GPOS side, there is a sampling library that contains a samples manager agent. When a new sample value arrives, the samples manager updates the corresponding data sample in local memory (e.g., S1 and S2 in Fig. 12a). The access to these local data samples is protected through a readers-writer lock which allows several consumers to access the same data sample concurrently.
5 Evaluation

This section evaluates the satisfaction of the qualitative and quantitative requirements in Table 1 on a physical implementation.

5.1 Evaluation environment

We implemented the dualoscom architecture and the traditional approach from Fig. 6a on top of SafeG; and evaluated them in this environment:

PB1176JZF-S board [29],
- 32KB Cache (flushed every measurement)
- DRAM: 128MB (Non-Secure memory)
- PSRAM: 8MB (Secure memory)

RTOS: TOPPERS/ASP v1.6.
GPOS: Linux v2.6.33 with buildroot [30].
VL: TOPPERS/SafeG v0.2.

5.2 Reliability evaluation

5.2.1 Memory isolation (req. 3.1a)

Memory isolation is satisfied by the dualoscom architecture thanks to the use of two levels of protection. In our implementation, the first level of protection is implemented through the permissions associated to a Linux device file (e.g., /dev/dualoscom) which are read/write for user tasks belonging to the comm group, and null for the rest of tasks. This device file is managed by the dualoscom Linux driver, and can be used for mapping the shared memory region into user space; or for the transmission of channel events. Protection with channel granularity can also be implemented at the cost of an increased memory usage due to the need of separated memory pages for each channel. The second level of protection is guaranteed by the TrustZone configuration as in the original architecture of SafeG.

5.2.2 Shared control data (req. 3.1b)

Dual-OS communication systems that follow the traditional approach depicted in Fig. 6a store all control data in trusted memory, where it is not accessible by the GPOS. In contrast, dualoscom stores control data in shared memory—which is accessible by untrusted-privileged software—and uses a four steps algorithm (see Section 4.1) involving a copy of the control data to RTOS memory. Since all updates to control data are done using validated

<table>
<thead>
<tr>
<th>function</th>
<th>copied bytes</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>block_alloc</td>
<td>0</td>
<td>0µs</td>
</tr>
<tr>
<td>block_free</td>
<td>0</td>
<td>0µs</td>
</tr>
<tr>
<td>block_getbuffer</td>
<td>0</td>
<td>0µs</td>
</tr>
<tr>
<td>block_enqueue</td>
<td>12 bytes</td>
<td>2.15µs</td>
</tr>
<tr>
<td>block_dequeue</td>
<td>4 bytes</td>
<td>0.75µs</td>
</tr>
<tr>
<td>event_send</td>
<td>0</td>
<td>0µs</td>
</tr>
<tr>
<td>event_wait</td>
<td>0</td>
<td>0µs</td>
</tr>
<tr>
<td>event_select</td>
<td>0</td>
<td>0µs</td>
</tr>
<tr>
<td>filter_set</td>
<td>0</td>
<td>0µs</td>
</tr>
</tbody>
</table>

Table 3 Tasks for the evaluation of the message interrupt rate limiting functionality.

<table>
<thead>
<tr>
<th>Task</th>
<th>OS</th>
<th>GP</th>
<th>C</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTH</td>
<td>RTOS</td>
<td>High</td>
<td>5ms</td>
<td>100ms</td>
</tr>
<tr>
<td>GPM</td>
<td>GPOS</td>
<td>Medium</td>
<td>10ms</td>
<td>100ms</td>
</tr>
<tr>
<td>RTL</td>
<td>RTOS</td>
<td>Low</td>
<td>40ms</td>
<td>100ms</td>
</tr>
</tbody>
</table>
The dualoscom system is configured with a single communication channel for the GPM task to send messages to the RTH task every 100 ms. The RTL task does not use the dualoscom system at all. Instead, it is activated every 100 ms, and at each activation it consumes 40 ms of execution time with a global priority lower than the priority of the GPM and RTH tasks. Under an ideal scenario (see Fig. 13a) the worst-case response time (WCRT) of the RTL task (i.e., the length of the longest interval from the task’s release till its completion) should always be shorter than $\sum C_i = 55 ms$. However, if the GPM task misbehaves—due to a bug or a malicious user—it can potentially cause the RTL task enter starvation by generating a message overload attack against the RTH task (see Fig. 13b). Note that the time for the GPM task to send a single message is an order of magnitude lower than its execution time budget. Fig. 13c illustrates how the dualoscom message interrupt rate limiting functionality can address this issue, by disabling the reception of message interrupts (e.g., using the RTOS interrupt controller) whenever the message interrupt rate limiting value is overrun.

We inserted malicious code into the GPM task for generating message overload attacks against the RTH task. Then, we measured the WCRT of the RTL task with and without the use of a bursty message interrupt rate limiter (note that the strict limiter is a particular case of the bursty limiter). Fig. 14 displays the measured WCRT of the RTL task for different malicious interrupt burst rates; and three different limiting values (i.e., 1, 5 and 10 interrupts/100ms). We observe that without a limiter, the WCRT of the RTL task increases proportionally to the malicious burst rate until it cannot meet its own deadlines for burst rates over 10. Then, it continues increasing with values close to the theoretical WCRT of the RTL task, which is defined by the smallest $x \in \mathbb{R}^+$ that satisfies the following formula [31] where $N$ represents the malicious burst rate:

$$x = C_{rtl} + \left\lceil \frac{x}{T_{gpm}} \right\rceil C_{gpm} + \left\lceil \frac{x}{T_{rth}} \right\rceil C_{rth} \cdot N$$

For malicious burst rates above 17 interrupts per 100ms, the RTL task enters starvation. This occurs because the utilization of the RTH and GPM tasks reaches 100% for burst rates of 18 or more (e.g., $\frac{5 \cdot 18 + 10}{100} = 1$). In contrast, when limiters are used the WCRT of the RTL task is always upper-bounded by substituting $N$ in Eq. 1 with the corresponding limiting value (i.e., 1, 5 and 10 interrupts/100ms).

For that reason, we can say that the dualoscom architecture satisfies requirement 3.1c.

5.2.4 Memory faults (req. 3.1d)

This requirement is satisfied because the region of shared memory used for communications is allocated statically during configuration time. In our implementation, we reserve the region of shared memory by using the `mem` Linux kernel boot parameter; and then create the necessary kernel page tables—in the dualoscom driver—using Linux `ioremap` functionality [32].

5.2.5 Unbounded blocking (req. 3.1e)

The dualoscom architecture has two functions that can block the calling task: `event_wait` and `event_select`. Both of them have a timeout pa-
rameter to protect RTOS tasks against the potential non-cooperation of the GPOS. The remainder interface uses non-blocking algorithms, and therefore we can say that requirement 3.1e is satisfied.

5.2.6 Code modifications (req. 3.1f)

Table 4 presents the number of added source lines of code (SLOC) and the binary size increase for our reference implementation. Requirement 3.1f is satisfied because neither the RTOS kernel nor the SafeG monitor required any modifications. The Linux kernel was extended with a simple character driver that supports the `mmap` operation—used at initialization—and the `ioctl` operation which is used for the transmission of events.

<table>
<thead>
<tr>
<th>OS</th>
<th>Level</th>
<th>SLOC</th>
<th>Binary increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP</td>
<td>user</td>
<td>530 lines</td>
<td>5980 bytes</td>
</tr>
<tr>
<td>ASP</td>
<td>kernel</td>
<td>0 lines</td>
<td>0 bytes</td>
</tr>
<tr>
<td>Linux</td>
<td>user</td>
<td>483 lines</td>
<td>5456 bytes</td>
</tr>
<tr>
<td>Linux</td>
<td>kernel</td>
<td>279 lines</td>
<td>280 bytes</td>
</tr>
<tr>
<td>SafeG</td>
<td>monitor</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5.3 Efficiency evaluation

5.3.1 Throughput (req. 3.2a)

We implemented the traditional approach in Fig. 6a on top of SafeG, and compared its performance against the dualoscom approach. Fig. 15 shows the results of the communications overhead of each approach (all measures are averaged). We observed that the dualoscom approach provides a constant overhead independent of the number of bytes being transmitted. This is a natural consequence of using zero-copy shared memory communications. In contrast, the overhead observed for the traditional approach is proportional to the amount of bytes being transmitted. The reason is that the main source of overhead in the traditional approach is caused by the number of unnecessary data copies—(1), (2) and (3) in Fig. 16—and context switches. We also observed that differences in the communication overhead are smaller when the amount of data being transmitted is below 128 bytes. In particular, we observed that the traditional approach performed better for 32 bytes blocks. However, the main reason for that is that the dualoscom measurements include synchronization overhead (e.g., event notifications), while in the traditional approach the Linux task just polls (i.e., busy-wait) for the arrival of new messages at a `SCHED_FIFO` priority (this is the typical implementation of the traditional approach[7]). If the dualoscom approach is used in polling mode, the overhead for the transmission of a 32 bytes block decreases to 30µs. This is smaller than the 47µs required by the traditional approach to transmit a
single byte. From these observations and other performance improvements—such as the filtering functionality or the ability to configure the synchronization needs of each individual channel—we can say that requirement 3.2a is satisfied.

5.3.2 Memory size (req. 3.2b)

The memory size overhead per channel caused by the shared control data used for communications is only $40 + 4 \cdot NB$ bytes, where $NB$ is the number of blocks in a channel. The memory overhead caused by the four steps algorithm (see Section 4.1) is 16 bytes/channel. Also, the architecture provides a configuration interface for minimizing the size of the shared memory required in a particular application. For all these reasons, we can say that requirement 3.2b is satisfied.

5.3.3 Interface (req. 3.2c)

Section 4.5 showed that the dualoscom interface is able to satisfy this requirement by supporting higher-level communication patterns—such as RPCs or unqueued messages—that are often used in embedded systems.

6 Conclusions

We proposed a new dual-OS communications architecture designed by taking into account the reliability and efficiency requirements of a dual-OS system. We implemented it and the traditional approach on a physical platform using a reliable dual-OS system (SafeG) that leverages ARM hardware security extensions for guaranteeing the RTOS reliability. Then, we evaluated both implementations through several experiments and observed that our architecture is more effective at minimizing the communications overhead and able to satisfy the reliability requirements of the RTOS.

Acknowledgments

We would like to thank the reviewers for their constructive comments. Part of this work is supported by the KAKENHI (23700035) and the Monbukagakusho scholarship.

References


