SafeG Dual-OS Communications Manual v.4.0

Daniel Sangorrín, Shinya Honda, Hiroaki Takada

SafeG 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. In this manual, we explain the SafeG dual-OS communications approach which is able to accomplish efficient communications without compromising the reliability of the RTOS.

1 Communications architecture

Here we describe the SafeG dual-OS communications (hereafter *dualoscom*) architecture.

1.1 Reliability properties

- *Memory isolation*: the dualoscom architecture relies on user-level shared memory for implementing dual-OS communications efficiently (see Fig. 1). In contrast to traditional approaches, 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. 1. 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*: 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 architecture.
- *Real-Time*: 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 FIFOs that exist in shared memory (see Fig. 2). The events path involves asynchronous notifications (implemented through inter-OS interrupts) and wait-event operations that may block the call-

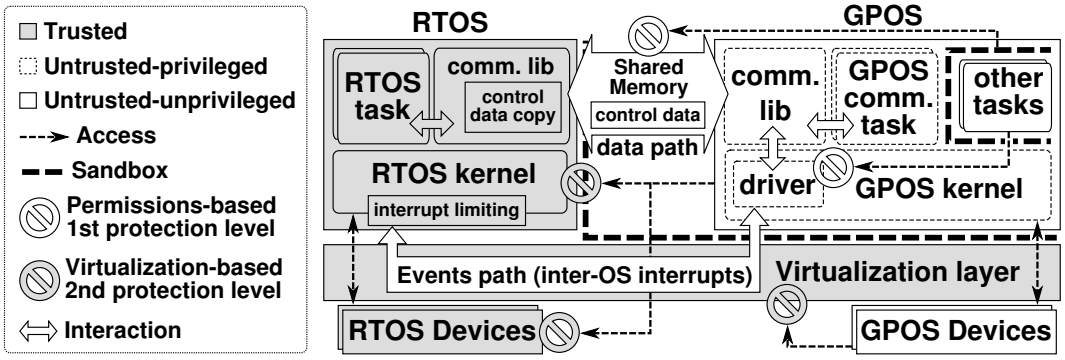


Fig. 1: The proposed dual-OS communications architecture.

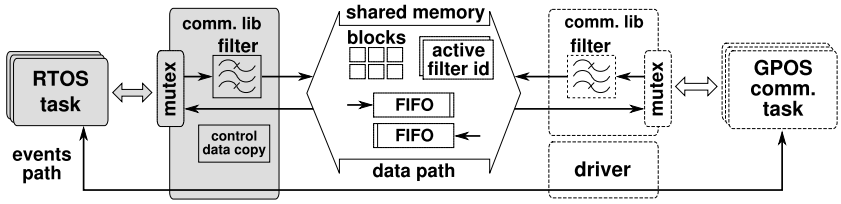


Fig. 2: Elements of a dualoscom communication channel.

ing 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. Finally, it is the responsibility of RTOS applications not to poll for new GPOS messages in an endless loop.

- *Memory faults*: 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*: 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*: 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.

1.2 Efficiency properties

- *Throughput*: 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

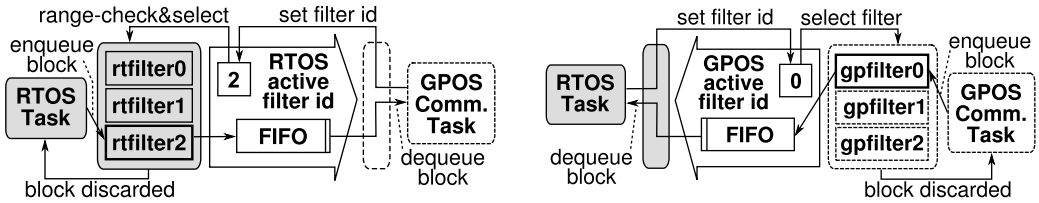
more mechanisms for reducing the overhead caused by unnecessary context switches: *filters* and *non-synchronized accesses*. Filters are functions that execute on the sender side of a channel (see Fig. 2 and Fig. 3) and are used for discarding the transmission of messages when they are not needed by 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*: 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*: 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.

1.3 Communication channels

A channel is a communication entity by means of which RTOS and GPOS untrusted-privileged tasks can exchange information. Fig. 2 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 $\Leftrightarrow$ 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. For example, in Fig. 4 a filter function (`tmp_update`) is used to discard the transmission of heater temperature values that do not represent updates of previous values. Fig. 3 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 $\Rightarrow$ 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 $\Rightarrow$ 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. 4), 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. 3a and 0 in Fig. 3b). While filter functions are located in the same memory region as the operating system where they execute, active fil-



(a) RTOS ⇒ GPOS communication direction (b) GPOS ⇒ RTOS communication direction

Fig. 3: Behavior of the filtering functionality in both communication directions.

ter 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. 3. Filter identifiers are automatically allocated by the dualoscom configurator tool during the configuration phase (see Fig. 4), 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. 3a, 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.

1.4 Dualoscom interface

1.4.1 The dualoscom build process

Fig. 4 illustrates the dualoscom build process through the heating devices example. As it is common practice in most RTOSs, 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 1.4.2. Then, the configuration file is parsed by the *dualoscom configurator* 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. 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 1.4.3 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.

1.4.2 Configuration interface

The configuration interface uses the next syntax:

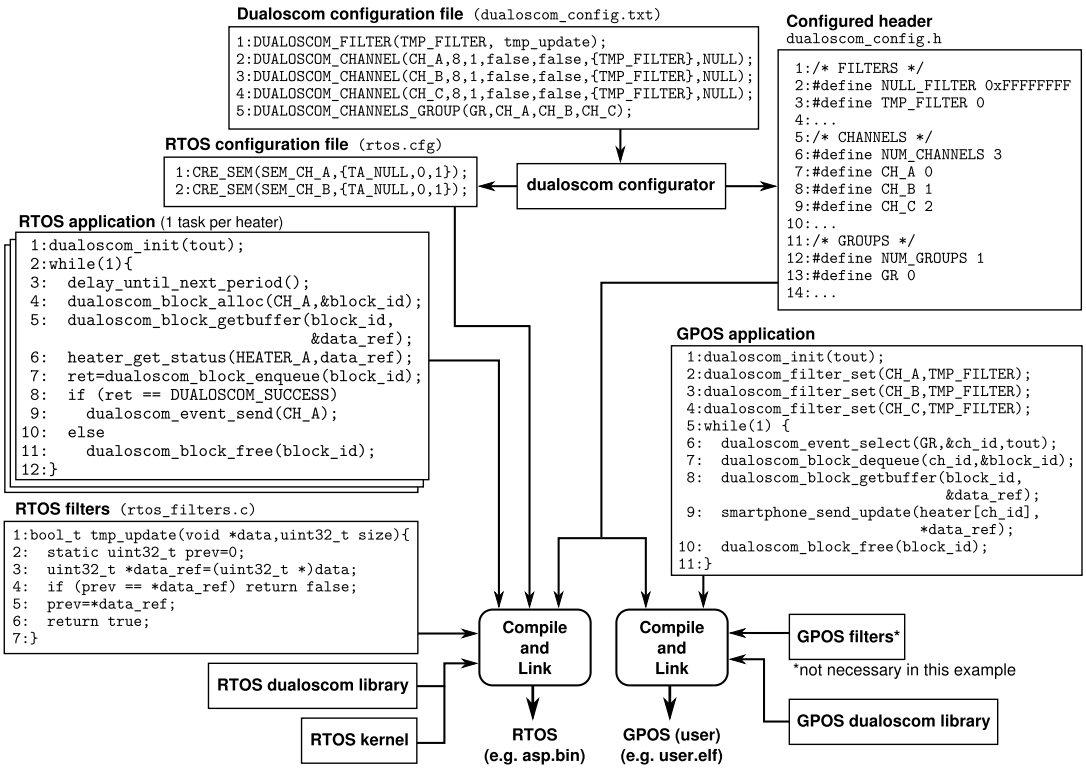


Fig. 4: The dualoscom build process.

`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. 4 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. 4) 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.

1.4.3 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. `DUALOSCOM_EMPTY`: no block is enqueued.
8. `DUALOSCOM_ALLOC`: 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(chan_id, block_id)`: releases a block back to the channel's pool where it belongs. May return errors 1, 2, 3, and 8.
- `block_getbuffer(chan_id, 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(chan_id, block_id)`: enqueues a block to a channel's FIFO. 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 errors 1, 2, 3, 7, and 8.

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.

1.5 Middleware

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

1.5.1 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. 5 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 chan-

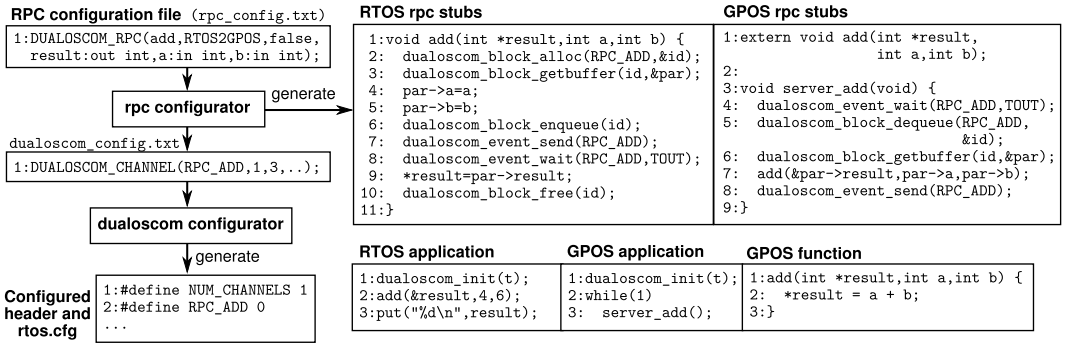
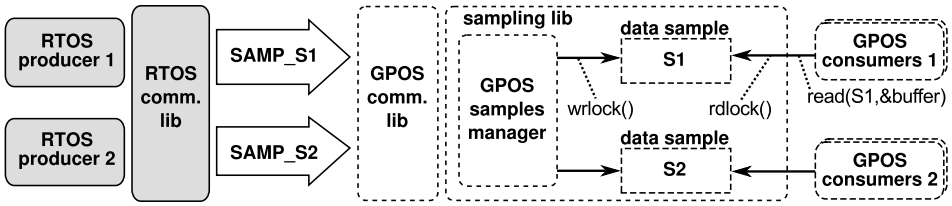
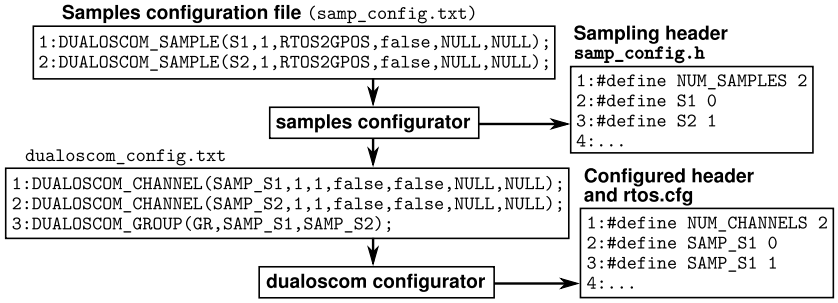


Fig. 5: Pseudocode of RPC communication.



(a) Unqueued/sampling messages on top of the dualoscom interface



(b) Configuration of unqueued/sampling mesages

Fig. 6: Support for unqueued/sampling messages on top of the dualoscom interface.

nels (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 configurator tool, which generates a dualoscom configuration file (e.g., `dualoscom_config.txt`) and the necessary stub functions on each OS that must also be linked into the final binaries.

1.5.2 Unqueued/Sampling messages

Unqueued messages—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. 6a 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 ⇒ 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:

`DUALOSCOM_SAMPLE()`: used to declare a data sample. The parameters are:

- `SAMPLE_NAME`: the name of the data sample. After configuration, the same name can be used to identify the data sample.
- `mx_size`: the maximum size of the sample.
- `direction`: indicates the communication direction (e.g., RTOS ⇒ 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. 6b). 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., `SAMP_S1` and `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. 6a). The access to these local data samples is protected through a readers-writer lock.